Finding a Shortest Walk Along a Sequence of Imprecise Regions in a Graph

by

Nima Sheibani

A thesis submitted to
The Faculty of Graduate Studies of
The University of Manitoba
in partial fulfillment of the requirements
of the degree of

Master of Science

Department of Computer Science
The University of Manitoba
Winnipeg, Manitoba, Canada
April 2019

Thesis advisor

Stephane Durocher

Author

Nima Sheibani

Finding a Shortest Walk Along a Sequence of Imprecise Regions in a Graph

Abstract

Given a sequence of disc-shaped imprecise regions, $R_1 \dots R_k$, and a graph G drawn in $\mathbb{R}^2$, we consider the problem of finding a shortest walk in G that visits the imprecise regions in order. Specifically, the walk starts at some vertex $v_1 \in V(G) \cap R_1$, follows a sequence of adjacent vertices from v_1 to some vertex $v_2 \in V(G) \cap R_2$, and so on, until reaching some vertex $v_k \in V(G) \cap R_k$. We prove that it is NP-hard to decide whether there is a weakly simple walk in G that visits the sequence of imprecise regions in order, answering a special case of an open problem posed by Löffler [51]. Moreover, we present an exact algorithm that finds a shortest walk visiting R_1 through R_k in any graph G, when the walk may include crossings. Generally, this algorithm runs in $O(kn^2 + APSP(n, m))$ time, but when the imprecise regions are mutually disjoint the running time reduces to $O(n^2 + APSP(n, m))$ where $APSP$ denotes the time complexity for all-pairs shortest path. Next we present an $O(kn)$ time algorithm that finds a weakly simple walk visiting R_1 through R_k in a graph G drawn in the plane, where G contains a non-crossing spanning tree, and the walk returned is not of minimum length in general.

Contents

List of Figures

Acknowledgments

First and foremost, I want to express my profound gratitude to my supervisor Dr. Stephane Durocher for his support, patience and wonderful guidance, his kindness and endless time he allocated to me. Steph helped me to get familiar with different problem-solving skills and he gave me a perfect insight to the field of computational geometry. He gave me freedom to work on whatever interested me and he always supported me. I want to say thanks to not only one of the best supervisors, but one of the best persons in my life who helped me to make the best experience of my life for more than two years.

I express my gratitude to my other thesis committee members, Dr. Avery Miller and Dr. Jason Morrison, for their support. Avery was always ready to help and he always allocates his valuable time to me. I want to thank all of the professors in the Computer Science department, specially Dr. Helen Cameron who always encouraged me and supported me. Moreover, I want to thank Dr. Ahmed Ashraf from ECE department of University of Manitoba and Dr. Debajyoti Mondal from University of Saskatchewan for their help and support. Also, I want to say thanks to all GADA lab members, specially Dr. Hamid Hoorfar, for their support and providing a friendly atmosphere in our cozy lab.

Besides Steph's financial supports, I would like to acknowledge the financial support from department of computer science who provided Guaranteed Funding Package (GFP), the Faculty of Graduate Studies at the University of Manitoba who provided University of Manitoba Graduate Fellowship (UMGF), Graduate Enhancement of Tri-Council Stipends (GETS), and International Graduate Student Entrance Scholarship (IGSES) to me and also the University of Manitoba Travel Award. These financial supports helped me to develop my knowledge without having any financial

concerns.

Last but not least, I express my heartfelt gratitude to my parents for their endless love and support throughout my life.

Chapter 1

Introduction

Tracking a mobile object has become a common task as a result of the prevalence of a wide multitude of GPS-enabled (Geographical Positioning Systems) and wirelessly connected mobile devices. Using GPS to track a mobile object and recording movement history is useful in various geographical and geometric applications. The precision of the location calculated by GPS depends on several factors. Sometimes, due to a lack of satellite coverage, poor antennae, or device limitations, the calculation is not sufficiently precise to compute a specific point for the location, and instead returns a region within which the object may be located. This estimate is often represented by a disc with a center point and radius, modelling, respectively, the center of the estimated location and the estimated range of possible error.

In Geographical Information Systems (GIS), geometric positional information is used as the input data and, depending on the specific required application, various algorithms can be applied to derive the desired output [51, 64, 66]. Most of these algorithms assume the input data is accurate, while in real-world situations, there is imprecision in the input data. Most of the time, the input data is obtained from a

sensor which is not perfectly precise. For example, the degree of accuracy of GPS can fluctuate, e.g., due to a lack of coverage in blind spots (e.g., in a tunnel or obstruction from nearby buildings). In this case, storing the exact movement history of a mobile object is impossible and there can be gaps in the recorded trajectory. In GIS-related problems, specifically, object tracking, a common problem is to reconstruct a mobile object's trajectory from partial GPS data, often consisting of a sequence of (possibly imprecise) recorded positions [67, 44, 68, 49, 13, 57]. This discrete sequence represents a continuous trajectory. Applications that requires a continuous trajectory must interpolate between the discrete data points. A natural strategy is to connect these by a shortest path. If the mobile agent's motion is restricted to a given environment, e.g., a vehicle driving on a road network, then the shortest path trajectory must be restricted to the environment (i.e., the mobile agent can only travel along roads); see Figure 1.1.

In this thesis, we study three different problems to find a walk through a given graph, connecting a given sequence of regions in different settings, when there is imprecision in the input data. Given a sequence of imprecise regions and a graph drawn in plane, the aim is to find a walk on the graph that visits the imprecise regions in order, if it exists, possibly minimizing the length of the walk or requiring that the walk be weakly simple (i.e., the walk does not cross itself).

1.1 Motivation

Spatio-temporal problems have long been an attractive and challenging topic in computational geometry. One important problem is reconstructing a trajectory that follows a given sequence of vertices in an embeded graph. This graph can be a map of

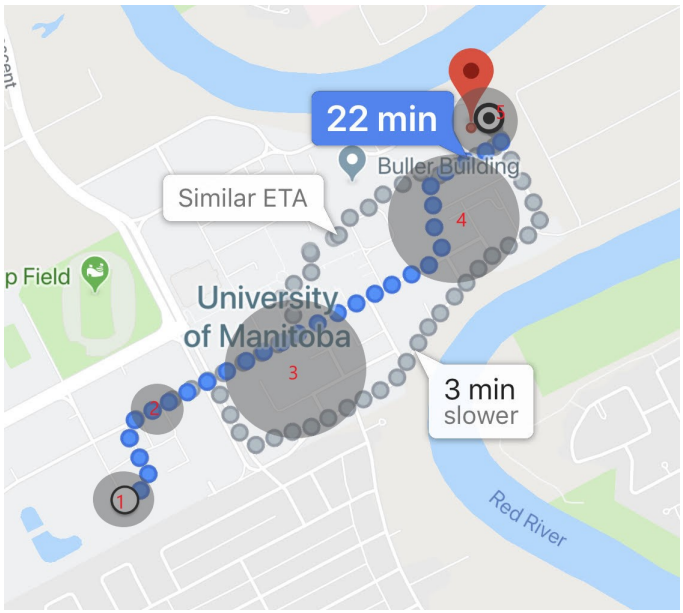


Figure 1.1: A part of Google map [1] which shows three possible ways for going from a building to a parking of University of Manitoba while passing through a recorded sequence of five imprecise regions (numbers denote the sequence of imprecise regions). There are three walks that connect five imprecise regions.

an actual road network on the ground (e.g., roads or railways) or a map that represents a network of underground pipes. Nowadays, some companies (e.g., [2, 3, 4, 5]) use a robot-assisted camera to inspect the condition of underground pipes. The robot starts from an access point, and its goal is to find a shortest path to another given point of the pipe network while conducting evaluations of the pipes (see Figure 1.2). Most of the time, the GPS on the robot shows where the robot is, but because it is underground, there are some blind spots which are not covered by GPS. In this case, the exact location of the robot is unknown, but during these short periods of time when there is no GPS coverage, the robot's location can be interpolated using prior (and subsequent) positions measured when GPS coverage was available. When GPS cannot precisely compute location, the robot's location can be approximated by a disc-shaped imprecise region. Usually, companies use wireless-controlled robots,

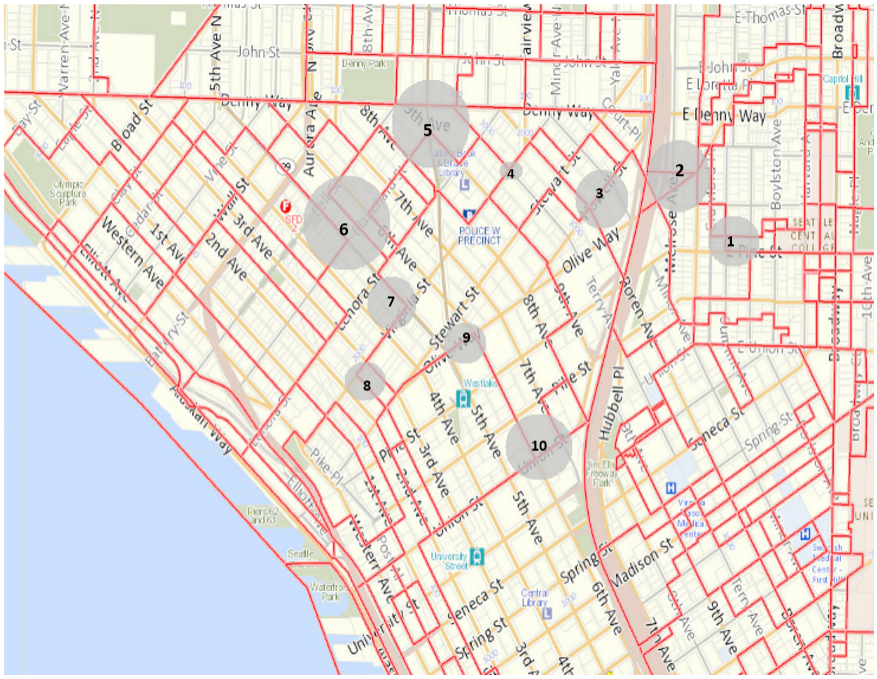


Figure 1.2: A part of Seattle water and sewer map [11]. The circles represent the imprecise regions. The sequence denotes the traveled trajectory of robot. The robot starts from a point in imprecise region number 1, check the status of a point in region 2, 3, . . .,9, and then it finishes its mission in region number 10.

but in some cases, the robot is controlled by a wire. In this case, crossing in the traveled path may cause a cut or entanglement in the wire. To reconstruct the robot's trajectory requires finding a non-crossing walk in the pipe network that connects a sequence of imprecise regions. (See definitions in Section 1.2.1).

Many health-related products such as Fitbit Ionic [6], Garmin GPS devices [7], or health-related mobile applications such as Apple Health [8], Strava [9], and Endomondo [10] allow users to record a history of their movement trajectory. Often, athletes want to keep track of their health data; particularly, they want to record a history of their walking or running trajectory (see Figure 1.3). Using a GPS-enabled device enables them to keep track of their health data, such as a map that tracks routes where they have walked. Similar to the previous scenario, sometimes limita-

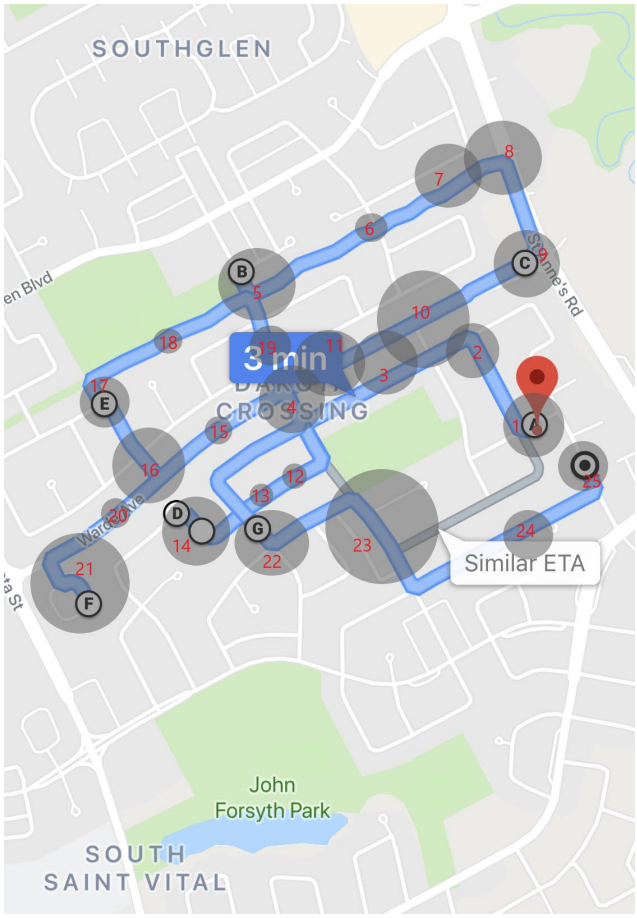


Figure 1.3: A part of Google map [1] which shows an athlete's traveled path. The athlete starts from point number 1, goes to points B, C, D, E, F, G, in order, and ends his walk in point number 25. The imprecise regions and the numbers on them show the sequence of traveled path. The walk may cross itself and the imprecise regions may overlap.

tions such as a lack of satellite coverage (e.g, because of skyscrapers in the city or poor coverage in mountains), or device working temperature (e.g., low temperature in winter), leads to gaps in the trajectory records. Displaying a continuous trajectory on a map requires connecting a given sequence of imprecise regions along a walk through a given road or trail network.

The idea of this thesis comes from an analysis of the above scenarios. In some cases, crossing in the traveled path is expensive or may cause some problems, whereas

in other cases a path may cross itself without causing any problem. To decrease expenses, it may be preferred to find a shortest path that visits the given sequence of imprecise regions.

1.2 Problem Definition

1.2.1 Models

The diversity of the applications in computational geometry leads to proposing different algorithms and various models for spatial data. In *the exact object model*, spatial objects are represented by points, lines, and regions for which exact coordinates are assumed to be known. In this model, regions are exactly bounded by polylines or curves which are called *boundaries* [30]. Many examples that suit this model are human-made, such as roads or railways. There are many spatial objects in the real world that do not have an exact boundary, or have a boundary which cannot be determined exactly. For example, consider a shoreline that varies depending on the season, recent rainfall, or the tide. These entities are denoted as *vague*, *indeterminate spatial objects*, [30] or *imprecise objects*. A region that indicates the estimated or possible location of an imprecise object is called an *imprecise region*; see Figure 1.6.

There are various types of graphs: A *general graph* is a combinatorial object which has vertices and edges, where each edge is a pair of vertices, but the graph's vertices and edges are not assigned positions in space. A graph G is *planar* if there exists a drawing of G in $\mathbb{R}^2$ in which each vertex is drawn as a point, each edge is drawn as a line segment joining the corresponding pair of vertices, and no two edges

cross. A *plane graph* is a planar graph with a given planar drawing in $\mathbb{R}^2$. Finally, a *Euclidean graph* is a weighted graph drawn in $\mathbb{R}^2$ in which edge weights are equal to the Euclidean lengths of the edges and vertices corresponds to points in $\mathbb{R}^2$. In this thesis we are interested in *Euclidean* and *plane* graphs. We typically denote a graph by G, its set of vertices by $V(G)$ and its set of edges by $E(G)$. We consider an *imprecise region* as a region in which a given object may lie whose precise position is unknown. We model it as a disc in the plane with a given center point c and radius r. The sequence $R_1, \ldots, R_k$ represents a sequence of disc-shaped imprecise regions. A *walk* on a given graph G is a sequence of vertices $v_1, \ldots, v_j$, where $v_j \in V(G)$ and for each j, $\{v_j, v_{j+1}\} \in E(G)$. In our problem we seek an *imprecise walk* across the regions $R_1, \ldots, R_k$ in G, that is, a walk that starts at some vertex $v_1 \in V(G) \cap R_1$, follows a sequence of adjacent vertices from v_1 to some vertex $v_2 \in V(G) \cap R_2$, and

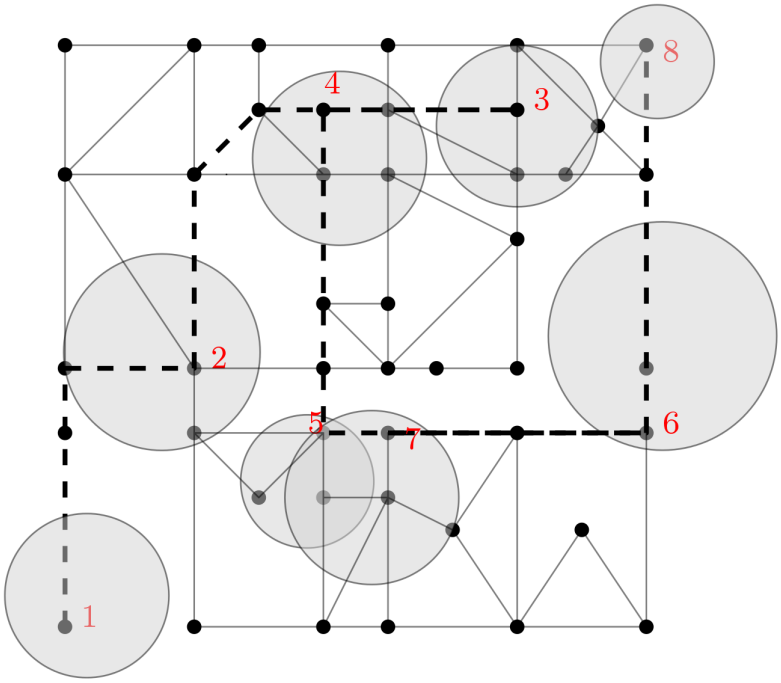


Figure 1.4: A *weakly simple walk* that visits all imprecise regions in order. Red numbers denote the order of visited vertices in the walk.

so on, until reaching some vertex $v_k \in V(G) \cap R_k$. Formally, a *walk* in the plane is a continuous function $P : [0, 1] \to \mathbb{R}^2$. "A walk is *simple* if it is injective, and it is *weakly simple* if for any $\epsilon > 0$, there is a simple walk whose Fréchet distance from P is at most ϵ" [16]. Therefore, a walk on a graph G drawn in $\mathbb{R}^2$ is *weakly simple* if the corresponding walk in $\mathbb{R}^2$ is weakly simple [16]; see Figure 1.4. "Two walks α and β in $\mathbb{R}^2$ *cross* if and only if there is an open neighborhood $A \subset \mathbb{R}^2$ that is homeomorphic to an open disc, such that $A \cap \alpha$ and $A \cap \beta$ are nonempty sub-walks of α and β, whose endpoints alternate around the boundary of A" [29]; see Figure 1.5.

1.2.2 Different Versions of The Problem

The following shows three different versions of the problems examined in this thesis.

We formally define Problem 1 as the following:

Instance:

- A graph G drawn in the plane,
- A sequence $R_1, \ldots, R_k$ of discs in $\mathbb{R}^2$ representing imprecise regions.

Problem 1. Find a *walk of minimum length* in G that visits one vertex contained in each of the imprecise regions $R_1, \ldots R_k$ in the specified order.

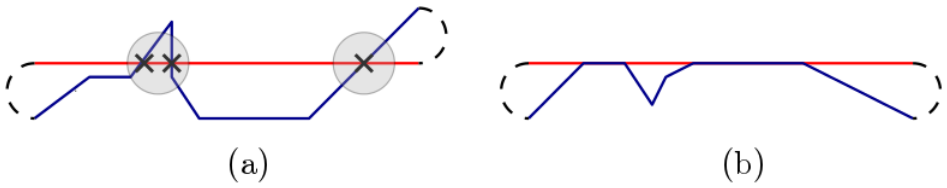


Figure 1.5: Blue and red lines are two parts of one walk. (a) Shows a crossing walk; the walk crosses itself in three points. (b) Shows a non-crossing (or *weakly simple*) walk; a *weakly simple walk* can touch itself, but cannot cross itself.

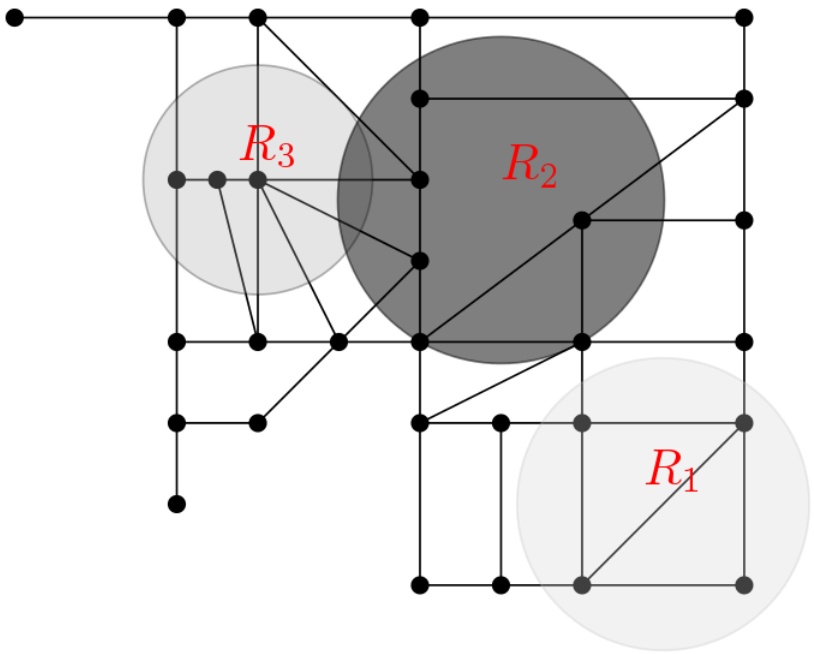


Figure 1.6: Three different imprecise regions in a graph drawn in the plane. The imprecise regions may overlap.

In addition to finding a walk that visits the imprecise regions in order, among all such walks we may wish to find a shortest walk. In the version of the problem defined above (Problem 1), crossing is allowed. Therefore, we want to find a shortest walk that may cross itself, and visits all of the imprecise regions in the specified order. The graph G in this version can be any arbitrary graph drawn in $\mathbb{R}^2$.

The next version of the problem can be formally defined as follows:

Problem 2. Find a *weakly simple walk* in G that visits one vertex contained in each of the imprecise regions $R_1, \ldots R_k$ in the specified order.

In this version of the problem, the walk does not necessarily have minimum length. We show that Problem 2 is NP-hard in general.

The following is a formal explanation of Problem 3:

Problem 3. Find a *weakly simple walk* in G that visits one vertex contained in

each of the imprecise regions $R_1, \dots R_k$ in the specified order, where G contains a non-crossing spanning tree.

The proposed algorithm for this version of the problem can be implemented on any non-negative weighted graph G drawn in the plane as long as there exists a non-crossing spanning tree T of G. We try to find a short walk, but the resulting walk may not be the walk with the minimum length, because being *weakly simple* (i.e., non-crossing walk) is the first priority in this version of the problem. This algorithm can be run in $O(kn)$.

1.3 Thesis Organization and Overview of Results

In this section we provide an overview of the results for the chapters in this thesis.

Chapter 1. This chapter is mainly an introduction to this thesis. Section 1.1 describes the motivation of this thesis. Section 1.2 includes an overview of our model (Section 1.2.1) and formal definition of different versions of the problem (Section 1.2.2). The last part of Chapter 1 is this section which introduces the organization of this thesis.

Chapter 2. In Chapter 2, we summarize some of important previous literature, mostly related to imprecise regions. In Section 2.1 some of the models for representing the imprecise regions are described briefly. Section 2.2 introduces the result of previous literature for finding a shortest non-crossing sequence in imprecise regions in the plane. Section 2.3 studies the algorithms for finding all-pairs shortest path in the weighted graphs and finally Section 2.4 includes the literature that focuses on finding shortest non-crossing path between terminal pairs.

Chapter 3. In this chapter we seek to solve Problem 1 (see Section 1.2.2). We

propose Algorithm 1 in Section 3.1, which solves Problem 1 exactly, when the walk may contain a crossing. We analyse the running time of Algorithm 1 in Section 3.2. Discussion and open problem of this chapter are described in Section 3.3.

Chapter 4. In Chapter 4 we seek to solve Problem 2 (see Section 1.2.2). We introduce different gadgets in Sections 4.1, 4.2 and 4.3 to show that finding a *weakly simple walk* that visits the imprecise region in order is NP-hard, even when the length of the walk is not required to be minimized. We transform an instance of PLANAR 3-SAT to an instance of our problem and show that it is NP-hard to decide whether a weakly simple walk exists. We describe the final construction in Section 4.4.

Chapter 5. This chapter studies Problem 3 (see Section 1.2.2). In this version of the problem we seek to find a weakly simple walk when the given non-negative weighted graph G drawn in $\mathbb{R}^2$ is restricted to graphs containing a non-crossing spanning tree. We introduce Algorithm 3 in Section 5.1 to solve Problem 3. We analyse the running time of Algorithm 3 in Section 5.2. We show that the Algorithm 3 runs in $O(kn)$ time. Discussion and open problems of this chapter are studied in Section 5.5.

Chapter 6. The final chapter describes conclusions and future work of the problems studied in this thesis.

Chapter 2

Related work

Various algorithmic problems have been studied previously under models of imprecise location. Depending on the problem and the nature of the uncertainty, some of these previous methods may be applicable to finding a solution to the problems examined in this thesis, as well as techniques used previously in shortest-path problems without uncertainty. In this chapter we provide brief descriptions of the models and results of previous articles.

2.1 Models for Representing Imprecise Regions

There are various models for imprecise regions, each of which has its advantages and disadvantages. Some studies have been conducted to deal with finding the proper model for the imprecise region. Montello et al. [53] asked people to identify specific neighbourhoods on a map to discover how ordinary people imagine an imprecise region and model it. Davies et al. [23] studied different models of imprecise regions to determine how these compare, and proposed that the best way to determine the

shape of an imprecise region depends on the type of region, the area of study, and the application. Common models to represent an imprecise region are fuzzy boundaries [37, 42, 18, 55], crisp boundaries [12, 56], and intermediate shapes such as the egg-yolk model [19, 24, 30, 38, 58]. de Berg et al. [24] described a new *Egg-yolk model*, in which the plane is partitioned into three regions: points that are certainly inside a given region, points that are potentially inside the region, and points that are certainly outside the region. Knauer et al. [46] suggested representing an imprecise region with a disc, specified by its center and radius. Löffler [51] proposed a model which represents an imprecise region with a square.

2.2 Shortest Non-crossing Sequence in Imprecise Regions in the Plane

An important question to address is to determine whether finding a shortest path among regions of uncertainty is NP-hard. Related to this question, Löffler [51] showed that given a sequence of imprecise regions, each of which is a square in the plane, it is NP-hard to decide whether a point can be placed in each of the given imprecise regions such that the resulting sequence of points forms a simple polygon [51]. He proposed that an imprecise region could be any arbitrary polygon and showed his method on square-shaped imprecise regions. Also, he showed that it is NP-hard to minimize the length of a simple tour that visits the regions with a specific order, when the connections between consecutive regions are not necessarily straight line segments [51]. To prove NP-hardness, he used a reduction from PLANAR 3-SAT. Moreover, he suggested two linear-time algorithms to compute the shortest and longest possible

tour for the case where the polygon is not required to be simple and the regions are squares [51].

2.3 Algorithms for All-Pairs Shortest Path in Weighted Graphs

There are two categories of problems for computing a shortest path in a graph: the first category is Single-Source Shortest Path (SSSP) [17, 22, 25, 26, 32, 33, 35, 34, 36, 41, 52, 65, 70], where the algorithm finds shortest path from a single-source vertex to all other vertices. The second category is All-Pairs Shortest Path (APSP), where the algorithm finds the shortest walk between all pairs of vertices, and finally finds the shortest walk in the whole graph, between the starting point and ending point. There are frequent articles that studied different algorithms to find All-Pairs Shortest Path (APSP) in real-weighted graphs. The classic Floyd-Warshall algorithm [20] runs in $O(n^3)$ for any real-weighted graphs, where n denotes the number of vertices. Fredman [31] was the first researcher who found a sub-cubic algorithm with a running time of $O(n^3 \log^{1/3} \log n / \log^{1/3} n)$. Other researchers [61, 27, 39, 62, 63, 71, 14, 40, 69] improved the running time of the algorithm for finding APSP. Chan [15] re-examined the all-pairs shortest path problem and presented a new algorithm with running time $O(n^3 \log^3 \log n / \log^2 n)$ and improved all previous algorithms for general real-weighted dense graphs. Moreover, for geometrically weighted graphs he used fast matrix multiplication to improve sub-cubic running time of the algorithm, where the weight of an edge is a function of the coordinates of its vertices.

For the single-source shortest path problem, Sedgewick and Vitter [59] attempted

to improve the running time of classical algorithms such as Dijkstra's algorithm by studying a heuristic algorithm for Euclidean graph, which finds a shortest path in a graph between two given vertices in $O(n)$ expected time on various classes of random graphs, although the worst-case time remains $O(m+n \log n)$-time, the worst-case time of Dijkstra's algorithm, where n shows the vertices and m represents the number of edges. The authors used the basic idea of Dijkstra's algorithm and then modified it. The idea of the modified algorithm is as follows: use Dijkstra's algorithm to compute the shortest path tree, but assign each vertex priority according to the distance in the tree to the start vertex plus the Euclidean distance to the destination vertex [59]. By using this heuristic, they could improve Dijkstra's algorithm running time in some cases.

2.4 Finding Shortest Non-Crossing Path Between Terminal Pairs

Lee [48] was the first researcher to introduce the non-crossing shortest path problem. Takahashi et al. [60] studied the problem of finding k non-crossing walks (i.e., the walks can overlap but cannot cross each other or themselves) in a graph G of minimum total length that connect all terminal pairs. The author described a divide-and-conquer algorithm to compute a non-crossing forest containing all the shortest paths between terminals. The algorithm runs in $O(n \log k)$ time.

Erickson and Nayyeri [29] generalized the Takahashi et al. result [60] for a special case and proposed an algorithm to find k non-crossing walks in G of minimum total length that connect all terminal pairs in $2^{O(h^2)} n \log k$ time where G is a n-vertex plane

graph with non-negative edge weights and k denotes the number of terminal pairs which are specified on h face boundaries. In this study, the walks between terminals may overlap, but they are not permitted to cross. The goal is to find an algorithm to compute a set of non-crossing shortest ST-walks (where for each i, the solution includes a walk from the starting terminal s_i to the ending terminal t_i, and no two walks cross) if it exists, or to report no set of ST-walks exists [29].

Papadopoulou [54] proposed an algorithm to compute the shortest path between k pair of points. As in the previous article, the paths are allowed to overlap, but not cross. The author considers the points in a polygon with one hole, and a pair of points may appear on the inner boundary (i.e., on the boundary of the hole) and/or outer boundary of the polygon. The proposed algorithm runs in $O(n + k)$ time.

Chapter 3

Exact Algorithm When Crossings Are Allowed

3.1 Algorithm Description

In this section, we propose an exact algorithm to find a walk of minimum length in G that visits the imprecise regions $R_1, \ldots R_k$ in the specified order when the walk may include crossings. Specifically, this section presents an algorithm that solves Problem 1 (see Section 1.2.2). In this version of the problem, the graph G can be any arbitrary graph drawn in $\mathbb{R}^2$ and each imprecise region R_i is any arbitrary region in the plane (e.g., a disc). For simplicity, we refer to $R_i \cap V(G)$ simply as R_i. The algorithm runs in $O(kn^2 + APSP(n, m))$ time, when m denotes the number of edges in G, n denotes the number of vertices in G, k denotes the number of imprecise regions, and $APSP(n, m)$ denotes the time required to compute an $[n \times n]$ all-pairs shortest distances matrix for G. When the imprecise regions are mutually disjoint, the running time reduces to $O(n^2 + APSP(n, m))$, which, for most graphs, corresponds to $O(APSP(n, m))$.

When implemented using Dijkstra's algorithm, $APSP(n, m) \in O(nm + n^2 \log n)$, which gives $O(n^2 \log n)$ time for planar graphs.

Theorem 3.1. *Given a graph G drawn in the plane with non-negative edge weights and a sequence $R_1, \ldots, R_k$ of imprecise regions, there exits an algorithm to find a walk that starts at some vertex $v_1 \in V(G) \cap R_1$, follows a sequence of adjacent vertices from v_1 to some vertex $v_2 \in V(G) \cap R_2$, and so on, until reaching some vertex $v_k \in V(G) \cap R_k$; the algorithm can be run $O(kn^2 + APSP(n, m))$ time, or $O(n^2 + APSP(n, m))$ time when $R_i \cap R_j = \varnothing$ for all $i \neq j$.*

Various algorithms can be used to compute the All-Pairs Shortest Paths (APSP). Our algorithm refers to an $[n \times n]$ matrix that stores the computed all-pairs shortest distances. The easiest solution is to compute a Single-Source Shortest Path (SSSP) tree n times (once per vertex) to find the APSP [21]. This algorithm can be run in $O(n^3)$ time [21]. We can improve the running time of the algorithm to $O(nm + n^2 \log n)$ when we implement Dijkstra's algorithm with Fibonacci heaps [21]. Another algorithm to find APSP is the Floyd-Warshall algorithm which runs in $\Theta(n^3)$ time [21]. Johnson's algorithm uses a reweighting technique to find APSP in $O(n^2 \log n + mn)$ time, which is faster than the Floyd-Warshall algorithm [21]. When the input graph G is planar, $m \in O(n)$, and the running time for both Dijkstra's and Johnson's algorithms decreases to $O(n^2 \log n)$. To distinguish its contribution to the running time of our algorithm, we denote the running time of the APSP algorithm that is used as $APSP(n, m)$.

Our algorithm is given as input a graph G drawn in the plane with non-negative edge weights and a sequence $R_1, \ldots, R_k$ of imprecise regions. The results in this chapter apply to arbitrary imprecise regions, i.e., when each imprecise region R_i is

an arbitrary subset of $V(G)$. Consequently, the results apply when imprecise regions are discs in the plane. Let $n = |V(G)|$ and let $m = |E(G)|$. In our algorithm, we want to find a shortest walk that starts at some vertex $v_1 \in V(G) \cap R_1$, follows a sequence of adjacent vertices from v_1 to some vertex $v_2 \in V(G) \cap R_2$, and so on, until reaching some vertex $v_k \in V(G) \cap R_k$. The algorithm works as follows: first, we compute the all-pairs shortest distances in G using an existing algorithm (e.g., Dijkstra's algorithm or the Floyd-Warshall algorithm). Generally, it suffices to use Dijkstra's algorithm when it is implemented with a Fibonacci heap. Let D denote an $[n \times n]$ matrix that stores the all-pairs shortest distances computed, where $D[u][v]$ denotes the length of a weighted shortest walk between the vertices u and v in G. Our algorithm constructs a $[k \times n]$ array A which stores a shortest walk through $R_1, R_2, \ldots, R_{i-1}$ to vertex $u \in R_i$; our pseudo-code (Algorithm 1) refers to $A[i][u]$ as the length of the corresponding walk. The values of A are initialized to ∞ and updated as shorter walks are discovered. We initialize the distance between the first vertex in R_1 and itself as 0, i.e., for all $v \in R_1 \cap V(G)$, $A[1][v] = 0$. The rest of the algorithm examines all pairs of the vertices in all pairs of consecutive imprecise regions in order. If it finds any shorter path it replaces the new walk with the previous

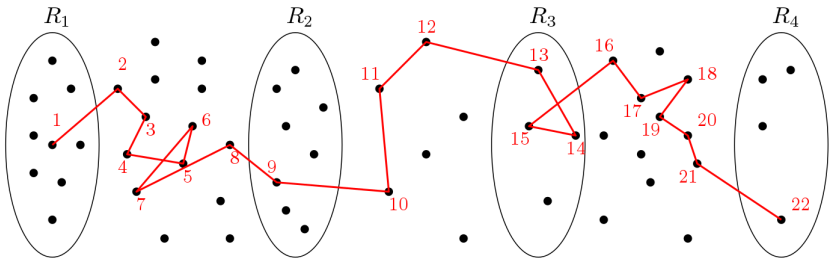


Figure 3.1: A part of graph G. In this figure we just show the imprecise regions, vertices, and the walk of minimum length that goes through imprecise regions. The walk starts from $v_1 \in V(G) \cap R_1$, continues its way to $v_9 \in V(G) \cap R_2$, $v_{13} \in V(G) \cap R_3$, and finally visits $v_{22} \in V(G) \cap R_4$. Crossing is allowed in Problem 1.

Algorithm 1 Finding a Shortest Walk

Compute all-pairs shortest distances, store in $D[0:n-1][0:n-1]$

for $i = 1$ *to* $k-1$ **do**

  for *all* $u \in V(G) \cap R_i$ **do**

    if $i = 1$ **then**

      $A[i][u] \leftarrow 0$

    end

    $A[i+1][u] \leftarrow \infty$

    for *all* $v \in V(G) \cap R_{i+1}$ **do**

      if $A[i][u] + D[u][v] < A[i+1][v]$ **then**

        $A[i+1][v] \leftarrow A[i][u] + D[u][v]$

      end

    end

  end

  if $i = k$-1 **then**

    $min \leftarrow \infty$

    for *all* $v \in V(G) \cap R_k$ **do**

      if $min > A[k][v]$ **then**

        $min \leftarrow A[k][v]$

      end

    end

  end

Return min {this is the length of the shortest walk}

one. The pseudo-code of our algorithm is presented in Algorithm 1.

3.2 Time Analysis

Our algorithm's running time is dominated by its first part, which is one of the existing algorithms for finding all-pairs shortest paths. Generally, it suffices to use Dijkstra's algorithm when it is implemented with a Fibonacci heap. This implementation leads to a running time of $O(mn + n^2 \log n)$ for computing the array D, which is $O(n^2 \log n)$ when the input graph G is planar. As can be seen in Algorithm 1, the second part of the algorithm can be run in $O(kn^2)$ time.

Lemma 3.2. *The worst-case running time of Algorithm 1 after computing the all-pairs shortest distances matrix is $O(kn^2)$, or $O(n^2)$ when the imprecise regions are mutually disjoint.*

Proof. When imprecise regions are mutually disjoint, we can bound the total running time by $\sum_{i=1}^{k-1} r_i r_{i+1}$, where $r_i = |R_i \cap V(G)|$ denotes the number of vertices in the ith imprecise region. Let n_i denote the total number of vertices in the first i imprecise regions.

Claim:

$$\forall j \geq 2, \sum_{i=1}^{j-1} r_i r_{i+1} \leq n_j^2. \tag{3.1}$$

Proof by induction on k.

Base case: Show (3.1) is true when $j = 2$.

$$r_1 r_2 < (r_1 + r_2)^2 = n_2^2.$$

Inductive hypothesis:

Choose any $k \geq 1$ and assume (3.1) is true when $j = k$:

$$\sum_{i=1}^{k-1} r_i r_{i+1} \leq n_k^2. \tag{3.2}$$

Inductive step: Show true when $j = k + 1$.

$$\sum_{i=1}^{k} r_i r_{i+1} = \left(\sum_{i=1}^{k-1} r_i r_{i+1} \right) + r_k r_{k+1}$$

$$\leq n_k^2 + r_k r_{k+1} \qquad \text{by (3.2)}$$

$$= \left(\sum_{i=1}^{k} r_i \right)^2 + r_k r_{k+1}$$

$$< \left(\sum_{i=1}^{k+1} r_i \right)^2.$$

$$= n_{k+1}^2.$$

When there is no restriction on the intersection of imprecise regions, the three nested loops give $O(kn^2)$ time. The algorithm's total time in this case is $O(kn^2 + APSP(n, m))$. Note that when imprecise regions are disjoint, the average number of vertices in each imprecise region is at most $\frac{n}{k}$, while when the imprecise regions are not disjoint it is $\Theta(n)$ in the worst case.

□

Theorem 3.3 follows from Lemma 3.2.

Theorem 3.3. *Algorithm 1 for finding a shortest walk between the imprecise regions in graph G drawn in $\mathbb{R}^2$ can be run in $O(kn^2 + APSP(n, m))$ time or $O(n^2 + APSP(n, m))$ time when the imprecise regions are mutually disjoint.*

Since $APSP(n, m)$ is typically $\Omega(n^2)$; it follows that the running time when imprecise regions are mutually disjoint is dominated by the computation of APSP, resulting in a running time of $O(APSP(n, m))$.

3.3 Discussion

To solve Problem 1 faster we can consider some assumptions (e.g., consider a fixed small number of vertices in each imprecise region). As we discussed above, the running time for our algorithm is $O(f(n, m)+APSP(n, m))$ when $f(n, m)$ denotes the running time for the second part (last three loops) of the Algorithm 1. It remains open whether there exists an algorithm that solves Problem 1 in $f(n, m) \in o(kn^2)$ time for general imprecise regions and $f(n, m) \in o(n^2)$ time when the imprecise regions are mutually disjoint. Finally, it might be possible to avoid computing all-pairs shortest paths, and to compute only a subset of these, thereby reducing the time complexity currently required by $APSP(n, m)$.

Chapter 4

Hardness Result

In this chapter we show that Problem 2 is NP-hard, even when the length of the walk is not required to be minimized. Given a sequence S of k imprecise regions (possibly overlapping) in an Euclidean graph G, we are looking for a weakly simple walk that starts at some vertex $v_1 \in V(G) \cap R_1$, follows a sequence of adjacent vertices from v_1 to some vertex $v_2 \in V(G) \cap R_2$, and so on, until reaching some vertex $v_k \in V(G) \cap R_k$; see Figure 1.4. Our reduction shows hardness when imprecise regions are discs in the plane; this implies hardness for arbitrarily shaped imprecise regions.

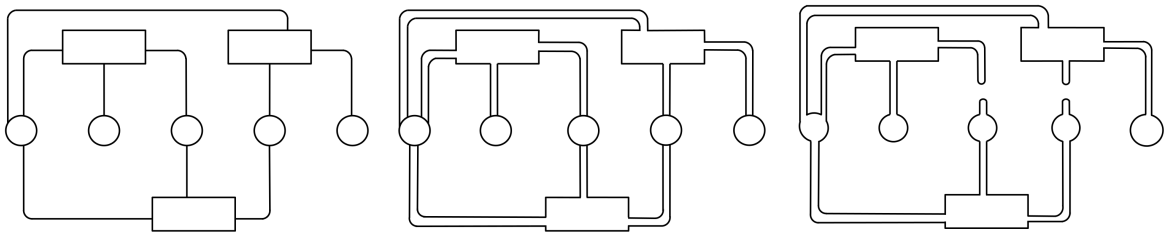


Figure 4.1: (a) An instance of PLANAR 3-SAT. (b) A weakly simple walk in the graph constructed after the initial transformation can include multiple cycles. (c) To prevent this, we introduce a gadget to cut cycles while maintaining the transmission of truth values on both sides of a cut. We represent variables by circles and clauses by rectangles.

The following describes the instance and the problem:

Instance:

- An Euclidean graph G drawn in the plane,
- A sequence R_1,..., R_k of discs in $\mathbb{R}^2$ representing imprecise regions.

Problem 2. Find a *weakly simple walk* in G that visits the imprecise regions $R_1, \ldots R_k$ in the specified order.

The solution to Problem 2 is a weakly simple walk that is not necessarily of minimum length. We show deciding whether G contains a weakly simple walk that visits $R_1, \ldots, R_k$ in order is NP-hard. We describe a reduction that transforms any instance of PLANAR 3-SAT into a graph drawn in the plane and a sequence of disc-shaped imprecise regions; our reduction is similar to that used by Löffler [51]. We show that a weakly simple walk exists in the transformed graph if and only if the PLANAR 3-SAT instance is satisfiable. We describe how to construct variable vertices, clause vertices, and edges connecting variables to the clauses in the transformed PLANAR 3-SAT instance. We use variable chains and clauses to make a tree-like structure, as can be seen in Figure 4.1(c).

Definition 4.1. PLANAR 3-SAT

Instance: An instance of *PLANAR 3-SAT* is a set of variables $x_1, \ldots, x_n$, each of which is assigned the truth value of *true* or *false*, and a set of clauses $c_1, \ldots, c_m$, where each clause is the disjunction of three literals, and each literal is a variable x_i or its negation $\overline{x_i}$. The set of variables and clauses define the vertices of a graph, and the inclusion of a variable x_i in a clause c_j defines an edge between the corresponding vertices. The resulting graph can be drawn in the plane such that the variable vertices

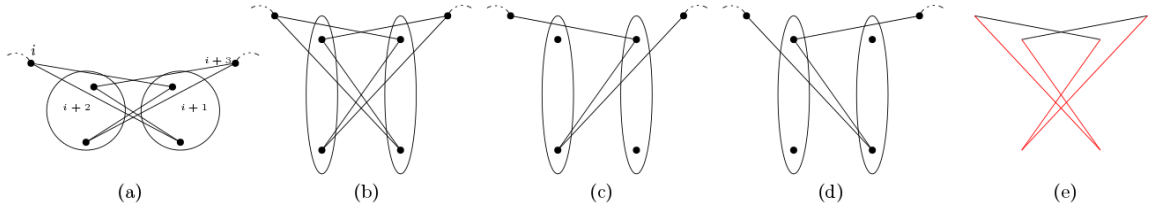


Figure 4.2: (a) A scissor gadget. Circles show the disc-shaped imprecise regions, points inside the imprecise regions are vertices. For simplicity, we show single-vertex imprecise regions as a vertex (without explicitly drawing an imprecise region around it). Imprecise regions are labelled by numbers denoting the order in which the walk must visit them in the scissor gadget. (b) We narrow the disc-shaped imprecise region to have a clear view of the scissor gadget. Only the illustration is transformed; the imprecise regions remain discs in the actual graph construction. (c) One of two possible solutions (possible weakly simple walks) that represents the *True* state, where positive legs are selected. (d) The other solution, representing the *False* state, where negative legs are selected. (e) Schematic representation for a scissor gadget, red lines show the *legs* of a scissor gadget and black lines show its *bases*.

are drawn along a line, with clause vertices drawn on either side, such that no two edges cross [47, 50, 28].

Problem: Does there exists a truth assignment (true of false) to the variables such that each clause is true, i.e., at least one of its literals (consisting of a variable or its negation) is true.

An instance of PLANAR 3-SAT can be seen in Figure 4.1. There exists a non-crossing embedding of G in the plane such that all literal vertices (set L) are positioned along the x-axis on a unit grid [47, 50, 28]. We assume that an instance of PLANAR 3-SAT refers to such an embedding; that is, the three literals of each clause are connected by three vertical line segments and at most three horizontal line segments [28].

4.1 Variables

In this section we describe gadgets for various parts of the transformation that correspond to variables, clauses, and wires (chains of scissor gadgets), as well as junction gadgets (for replicating the truth value of a variable across multiple wires) and transmission gadgets (for breaking loops to form a single walk).

4.1.1 Scissor Gadget

Choose any instance I of PLANAR 3-SAT. Let $x_1, \ldots, x_n$ denote the boolean variables of I and $c_1, \ldots, c_m$ denote its clauses. We represent each boolean variable x_i by a chain of *scissor gadgets* (which we call *wires*), each of which can be in one of two states, corresponding to the truth values *true* or *false*. A scissor gadget is a construction of two imprecise regions each of which has two vertices inside it (dual-vertex imprecise region) and another two imprecise regions with one vertex inside each of them (single-vertex imprecise region) that should be visited in the given order.

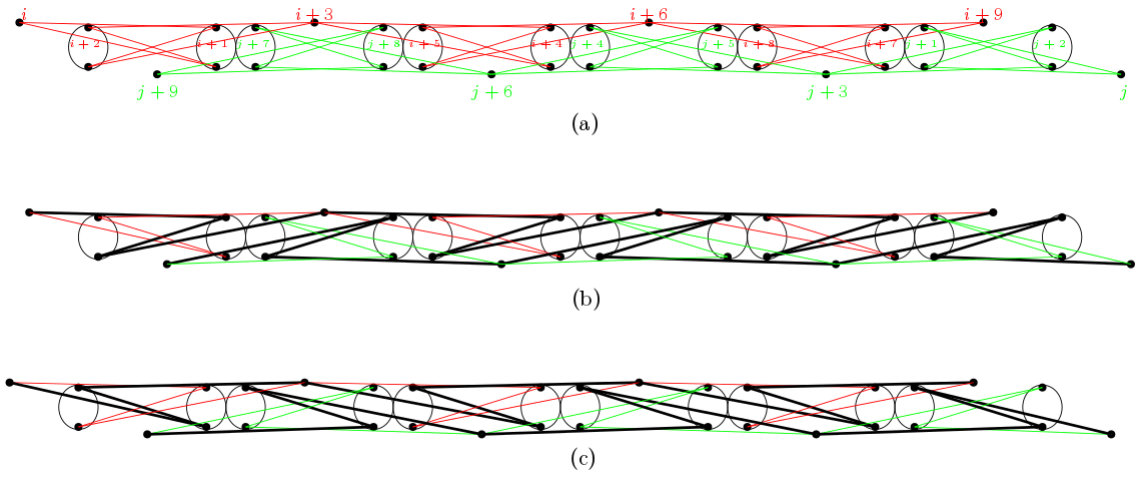


Figure 4.3: (a) A wire consisting of a sequence of scissor gadgets (b) *True* state non-crossing walk (c) *False* state non-crossing walk. Black lines show the possible non-crossing walk. The walk continues after $j + 9$ and $i + 9$, and before j and i.

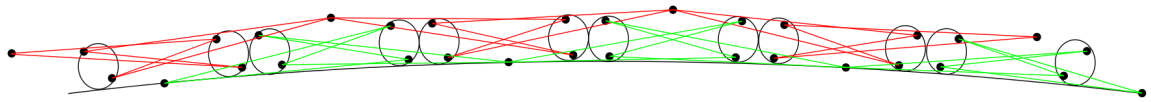


Figure 4.4: To make the final construction we can easily bend a chain of scissor gadgets to any angle, particularly for a 90 degree bend.

Each scissor gadget has six edges. Two edges that connect single-vertex imprecise region R_i and top vertex of imprecise region R_{i+2} to the bottom vertex of imprecise region R_{i+1} and another two edges that connect single-vertex imprecise region R_{i+3} and top vertex of imprecise region R_{i+1} to the bottom vertex of imprecise region R_{i+2} are called *legs*. The other two edges that connects single-vertex imprecise region R_i to the top vertex of imprecise region R_{i+1} and single-vertex imprecise region R_{i+3} to the top vertex of imprecise region R_{i+2} are called *bases*. We describe coordinates for a single scissor gadget; a variable consists of a chain of translated copies of a single scissor gadget that transmit the scissor gadget state, i.e., they form a wire; see Figure 4.2. Position two vertices in each of the dual-vertex imprecise region, one on the top, and another one on the bottom. Also, let r denote the radius and c denote a point center for the disc-shaped imprecise regions. We define coordinates and ordering (where the index i denotes the rank in the complete sequence of imprecise regions, to which each scissor gadget contributes four regions) for the components of a scissor gadget as the following:

1: A disc-shaped single-vertex imprecise region (R_i) with its vertex at (0, 4) and radius $r = \epsilon$, point center $c = (0, 4)$.

2: A disc-shaped dual-vertex imprecise region (R_{i+1}) with its vertices at (8, 0), (8, 2) and radius $r = 2+\epsilon$, point center $c = (8, 1)$.

3: A disc-shaped dual-vertex imprecise region (R_{i+2}) with its vertices at (3, 0),

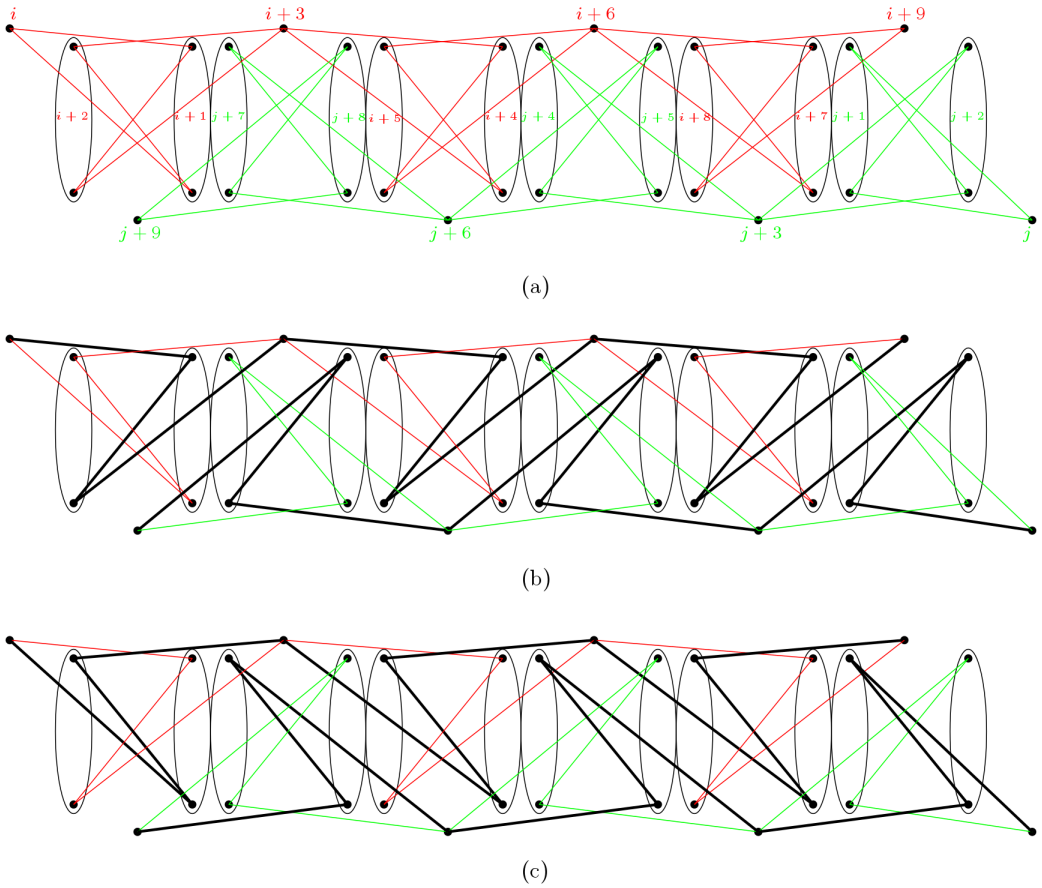


Figure 4.5: We re-scale Figure 4.3 to make it more clear. (a) A wire consisting of a sequence of scissor gadgets (b) *True* state non-crossing walk (c) *False* state non-crossing walk. Black lines show the possible non-crossing walk.

(3, 2) and radius $r = 2+\epsilon$, point center $c = (3, 1)$.

4: A disc-shaped single-vertex imprecise region (R_{i+3}) with its vertex at (11, 4) and radius $r = \epsilon$, point center $c = (11, 4)$.

Figure 4.2(a) shows this construction. Black points denote the vertices and discs represent the imprecise regions.

Any weakly simple walk must pass through both single-vertex imprecise regions and one of the vertices in each of the dual-vertex imprecise regions (the top vertex or the bottom vertex). As we show in Lemma 4.1, there are only two possible ways to make a *weakly simple walk* through a scissor gadget.

Lemma 4.1. *There are only two possible solutions to make a weakly simple walk through a scissor gadget.*

Proof. In a scissor gadget, the walk enters the gadget at the single-vertex imprecise region R_i. The walk has two possible edges to follow out of R_i, each leading to R_{i+1}. Each vertex in R_{i+1} has degree two, meaning that the walk must exit that vertex on the second edge. It is the same for imprecise regions R_{i+2}. One possible solution is to start from the leftmost single-vertex imprecise region R_i, then use the top vertex of the imprecise region R_{i+1}, then the bottom vertex of the imprecise region R_{i+2}, and continue the walk to the rightmost single-vertex imprecise region R_{i+3}. We consider this solution as the *true* state. Consider that if the walk follows out R_{i+1} to the top vertex in R_{i+2}, then when it follows out of R_{i+2} to R_{i+3}, there will be a crossing in the walk. The other solution that cause a crossing is to connect the vertex in single-vertex imprecise region R_i to the top vertex in R_{i+1}, and follow out to the top vertex in R_{i+2}; when the walk follows out of R_{i+2} to R_{i+3} there will be a crossing in the walk. Therefore, there is just one possible solution for *true* state. As mentioned above, the vertex in R_i has degree two, so another possible solution is to use the other edge of the vertex in R_i which connects the vertex in R_i to the bottom vertex of R_{i+1}. Because the degree of the vertex in R_{i+1} is two, the walk uses the other edge to visit the top vertex in imprecise region R_{i+2}. Then the walk follows out of R_{i+2} to the single-vertex imprecise region R_{i+3}. It is the only possible solution for the *false* state. Consider that if a walk follows out of the bottom vertex in R_{i+1} to the bottom vertex of R_{i+2}, when the walk wants to visit the single-vertex R_{i+3}, there will be a crossing in the walk. See Figure 4.2(c) and 4.2(d). Therefore, if we choose any walk other than these two possible solutions, there will be a crossing in the walk. To have

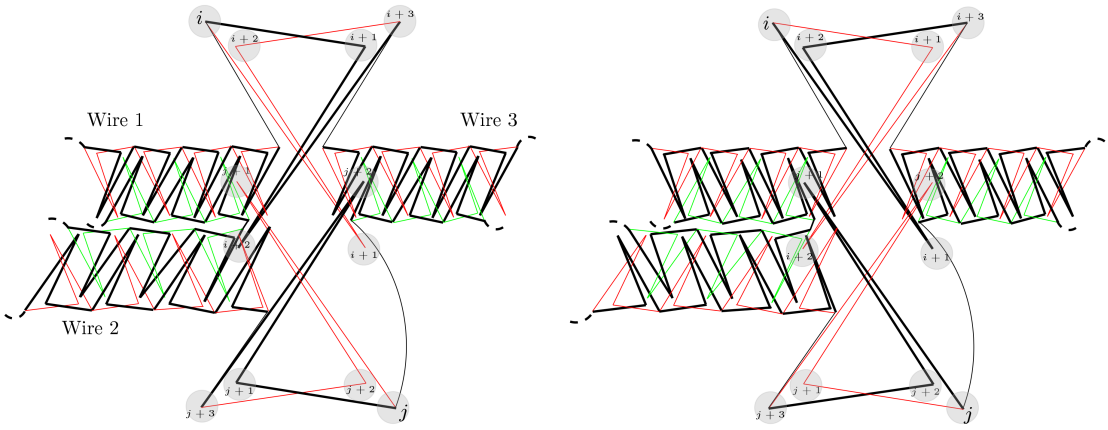


Figure 4.6: (a) A junction gadget copies the truth value of chains; this shows *true* state. (b) The other solution which shows *false* state. Grey circles represent the imprecise region. To make the figure more clear, instead of drawing the whole imprecise regions, we only show each vertex of the imprecise regions. The imprecise regions are indicated by a gray circle and the order of the them are denoted by black numbers.

a valid input to the PLANAR 3-SAT instance, we must have a *weakly simple walk* through the whole construction. Therefore, there are only two possible solutions for visiting a scissor gadget in the specified order. □

Lemma 4.1 can be followed by the following lemma:

Lemma 4.2. *There are only two possible solutions to make a weakly simple walk in a chain of scissor gadgets.*

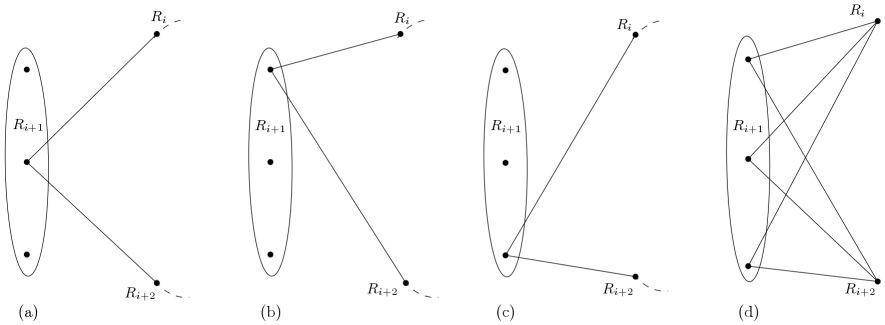


Figure 4.7: (a) The first solution. (b) The second solution. (c) The third solution. (d) Schematic representation for a clause gadget.

Proof. We enumerate all the cases that may happen when two scissor gadgets are linked. We consider only two scissor gadgets and for the other scissor gadgets that can be connected to these two, the proof is the same.

Case 1. Suppose that the scissor gadgets are not in the same state.

Case 1.1. Suppose the first scissor gadget is true and the second scissor gadget is false. In this case, the positive leg of the first scissor gadget and the negative leg of the second scissor gadget are selected. Therefore, the positive leg of the first scissor gadget crosses the negative leg of the second scissor gadget. Consequently, these scissor gadgets cannot be in different states. Thus, this case cannot occur.

Case 1.2. Suppose the first scissor gadget is false and the second scissor gadget is true. In this case, the negative leg of the first scissor gadget and the positive leg of the second scissor gadget are selected. Therefore, the negative leg of the first scissor gadget crosses the positive leg of the second scissor gadget. Consequently, these scissor gadgets cannot be in the different states. Thus, this case cannot occur. Therefore, in the above cases there is no weakly simple walk which joins the sequence of imprecise regions in order.

Case 2. Suppose both scissor gadgets have the same value (i.e., they are in the same state).

Case 2.1. Suppose they are both true. In this case, the positive leg of both scissor gadgets are selected and there is no crossing in the selected legs.

Case 2.2. Suppose they are both false. In this case, the negative leg of both scissor gadgets are selected and just as in Case 2.1. No crossings exist in the selected legs.

For having a satisfiable instance to our problem, we need a weakly simple walk. Therefore, there is only two possible solution when two or more scissor gadgets are

linked. □

4.1.2 Junction Gadget

We can *link* the scissor gadgets together. Each chain of scissor gadgets forms a wire that propagates the truth value of a variable. In this case, they must be in the same state, otherwise, there will be crossing in the walk. Consequently, a weakly simple walk may use either positive leg of the scissor gadgets or their negative leg, as can be seen in Figure 4.2(b) and 4.2(c). Figure 4.3 and 4.5 show an example of linked scissor gadgets. Therefore, if a scissor gadget needs to be connected rightward, the rightmost single-vertex imprecise region of the first scissor gadget must be connected to the leftmost single-vertex imprecise region of the second scissor gadget, see Figure 4.5(a). Making chains of scissor gadgets is possible if all of them are in the same state; i.e., the solution walk must follow the positive legs in each of the scissor gadgets or the negative legs, otherwise there will be a crossing in the walk. Therefore, there are only two possible solutions to make a chain of scissor gadgets. When the solution walk follows a given leg, we say it is the *selected leg*. The same variable can appear in multiple clauses. In our reduction, this requires replicating a variable's truth value onto multiple wires, each of which is a chain of scissor gadgets.

We can connect three chains of scissor gadgets together by using a *Junction* gadget to replicate the variable's truth value consistently across all branches. In this case, we need a wire to connect the junction gadget to two scissor gadgets on its sides, as it is shown in Figure 4.6(a) and 4.6(b). Consider that there should be a distance more than radius r of the largest imprecise region in the scissor gadget (i.e., greater than

$2 + \epsilon$) between the endpoint of linked scissor gadgets legs and their bases; otherwise, there is no possible solution (i.e., there is a crossing in the walk).

Lemma 4.2 can be followed by Lemma 4.3:

Lemma 4.3. *A sequence of imprecise regions in a junction gadget has a weakly simple walk if and only if all linked chains of scissor gadgets have the same state (i.e., the same truth value).*

Proof. We consider the top left chain of the junction gadgets as Wire 1, the bottom left chain as Wire 2, and the right chain of the junction gadgets as Wire 3, as can be seen in Figure 4.6. The following shows the different possible cases:
Case 1. Suppose two wires have different truth values.

Case 1.1. Suppose Wire 1 is true, Wire 2 is false and Wire 3 is false. In this case, the positive leg of Wire 1 is selected which implies that the positive leg of the junction gadget must be selected (based on Lemma 4.2). In Wire 2, the negative leg is selected, so the negative leg of Wire 2 crosses the positive leg of the junction gadget. Also, in Wire 3 the negative leg is selected. Therefore, there will be another crossing with the positive leg of the junction gadget. Consequently, Wire 2 and 3 cannot be in the false state, deriving a contradiction. Thus, this case cannot occur.

Case 1.2. Suppose Wire 1 is false, Wire 2 is true and Wire 3 is false. In this case, the negative leg of Wire 1 is selected which implies that the negative leg of the junction gadget must be selected. In Wire 2, the positive leg is selected, so the positive leg of Wire 2 crosses the negative leg of junction gadget. Consequently, Wire 2 cannot be in the true state, deriving a contradiction. Thus, this case cannot occur.

Case 1.3. Suppose Wire 1 is false, Wire 2 is false and Wire 3 is true. A proof analogous to that used in Case 1.1 shows that this case cannot occur.

Case 1.4. Suppose Wire 1 is false, Wire 2 is true and Wire 3 is true. In this case, the negative leg of Wire 1 is selected, implying that the negative leg of the junction gadget must be selected. In Wire 2, the positive leg is selected, so the positive leg of Wire 2 crosses the negative leg of junction gadget. Also, in Wire 3 the positive leg is selected. Therefore, there will be an other crossing with the negative leg of the junction gadget. Consequently, Wire 2 and 3 cannot be in the false state, deriving a contradiction. Thus, this case cannot occur.

Case 1.5. Suppose Wire 1 is true, Wire 2 is false and Wire 3 is true. In this case, the positive leg of Wire 1 is selected which implies that the positive leg of the junction gadget must be selected. In Wire 2, the negative leg is selected, so the negative leg of Wire 2 crosses the positive leg of junction gadget. Consequently, Wire 2 cannot be in the false state, deriving a contradiction. Thus, this case cannot occur.

Case 1.6. Suppose Wire 1 is true, Wire 2 is true and Wire 3 is false. A proof analogous to that used in Case 1.4 shows that this case cannot occur.

Therefore, in the above cases there is no weakly simple walk that joins the sequence of imprecise regions in order.

Case 2. Suppose all wires have the same truth value.

Case 2.1. Suppose Wire 1 is true, Wire 2 is true and Wire 3 is true (i.e, all wires are true). In this case, the positive legs of the wires and junction gadget are selected and none of the wires legs or junction gadget legs cross. Also, no crossings exist between the legs of different wires. Consequently, the walk may use wires and junction gadget without any crossing.

Case 2.2. Suppose Wire 1 is false, Wire 2 is false and Wire 3 is false (i.e, all wires are false). An argument analogous to that used in Case 2.1 shows that there will not

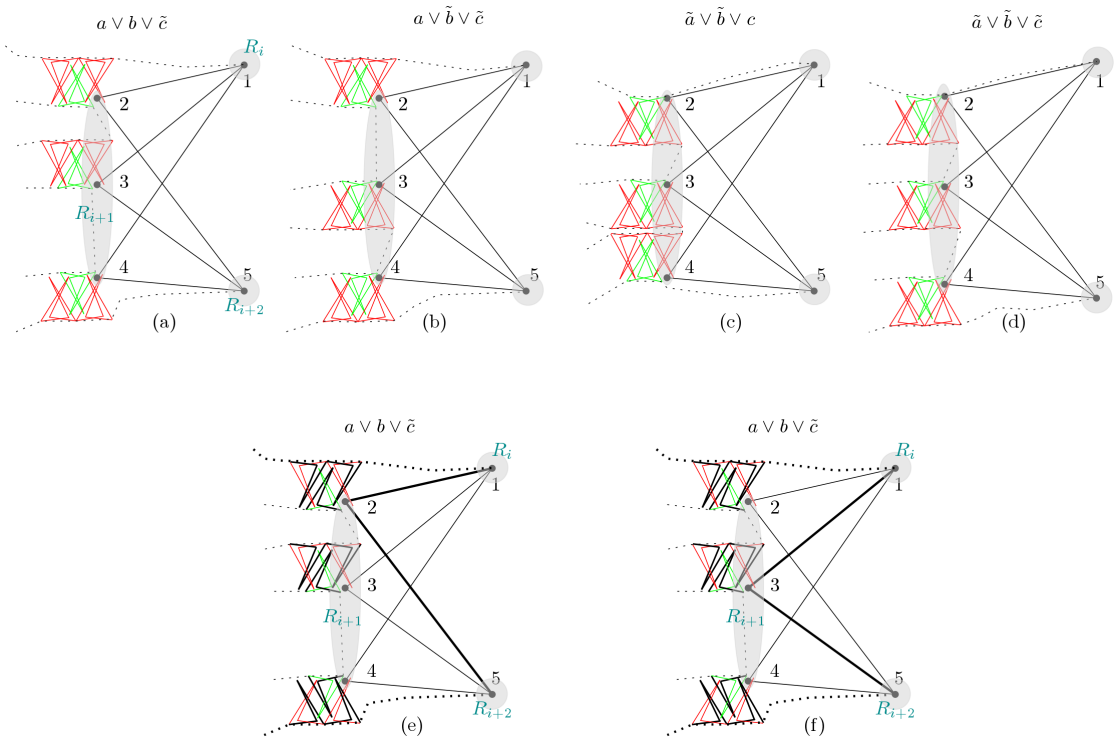


Figure 4.8: Four examples of different ways that chains of scissor gadgets can be connected to a clause gadget. There must be at least one possible way to visit the clause gadget, otherwise there will not be any possible solution to Problem 2. (e) is a one of the possible solutions to (a). (f) the other possible solution to (a). Vertex number 1 is in R_i, vertices number 2, 3, and 4 are in R_{i+1}, and vertex number 3 is in R_{i+2}.

be any crossing the in the walk which goes through the base and legs of wires and the junction gadget.

Therefore, a weakly simple walk joins the sequence of imprecise regions in order for case 2.1 and 2.2 where all of the wires and junction gadget have the same truth value. □

4.2 Clauses

We represent each clause by a *clause gadget*. A clause gadget is a construction of one imprecise region with three vertices inside it (multiple-vertex imprecise region) each of which is placed on the top, middle and bottom of the multiple-vertex imprecise region and two other single-vertex imprecise regions. We describe coordinates for a single clause gadget; other clause gadgets are translated and rotated copies of a single clause gadget. The following shows the coordinate and ordering (where i denotes the ordering) for the components of a clause gadget:

1: A disc-shaped single-vertex imprecise region (i) with its vertex at (4, 4) and radius $r = \epsilon$, point center $c = (4, 4)$.

2: A disc-shaped multiple-vertex imprecise region ($i + 1$) with its vertices at (1, 1), (1, 2), (1, 3) and radius $r = 2+\epsilon$, point center $c = (1, 2)$.

3: A disc-shaped single-vertex imprecise region ($i + 2$) with its vertex at (4, 0) and radius $r = \epsilon$, and point vertex $c = (4, 0)$.

A clause gadget has six edges that three of them connect single vertex imprecise region R_i to all of three vertices of multiple-vertex imprecise region R_{i+1} and the other three edges connect single vertex imprecise region R_{i+3} to all of three vertices of multiple-vertex imprecise region R_{i+1}. Three chains visit the clause gadget and at least one of them should be true, if the PLANAR 3SAT is a yes-instance. A *normal* literal connects to the clause gadget in the regular way (i.e., bases on top and legs on the bottom), while *negated* literals connect to clause gadget when they are mirrored horizontally. The main walk must use two single-vertex imprecise region (imprecise regions R_i and R_{i+2}) and one of the vertices in the multiple-vertex imprecise region R_{i+1}. There are three different ways that the main walk can visit the clause gadget.

The main walk enters the gadget from single-vertex imprecise region R_i, then visits one of the vertices in multiple-vertex imprecise region R_{i+1} (which leads to three different solution since there are three vertices in R_{i+1}) and then follows out the the gadget via the single-vertex imprecise region R_{i+2}, see Figure 4.7. In this case, at most two chains of scissor gadgets can block the clause gadget; otherwise, there will be no valid solution to Problem 2. For example, in Figure 4.8(a) there is a valid solution to the gadget in all cases except when literals *a, b,* and *c* are false, false, and true, respectively.

Lemma 4.4. *The clause gadget can be visited by a weakly simple walk if and only if at least one of the three literals is true.*

Proof. Here we enumerate all the cases that may happen when the main walk visits a clause gadget. We consider an order for the vertices to make the proof more clear, as shown in Figure 4.8.

Case 1. Suppose all literals are false. In this case all literals block the associated edges with Vertices 2, 3, and 4. Therefore, the main walk cannot visit the clause gadget. In this case, there is no valid solution to Problem 2.

Case 2. In the second case, only the top literal is true, i.e., the other two literals block associated edges with Vertices 3 and 4. Therefore, the main walk visits clause gadget by using Vertex 2 in R_{i+1}.

Case 3. Suppose only the middle literal is true, i.e., the associated edges with Vertices 2 and 4 in R_{i+1} are blocked. In this case, the main walk must visit R_{i+1} by using the Vertex 3.

Case 4. Suppose only the bottom literal is true (analogous to Case 2), i.e., associated edges with Vertex 2 and Vertex 3 are blocked. In this case, the main walk must visit

R_{i+1} by using Vertex 3.

Case 5. In the fifth case, if only the top literal is false then only edges associated with Vertex 2 in R_{i+1} are blocked. So, there are two possible solutions to visit the clause gadget; the main walk may use either Vertex 3 or Vertex 4 in R_{i+1}.

Case 6. Only the middle literal is false. Therefore, there will be two possible solutions; the main walk may visit the clause gadget by using associated edges with Vertex 2 or Vertex 4 in R_{i+1}.

Case 7. In this case, only the bottom literal is false (analogous to Case 5), so same as Case 5 and Case 6, there are two possible solutions. The main walk can use either edges associated with Vertex 2 or Vertex 3 to visit R_{i+1}.

Case 8. In this case all of three literals are true, i.e., non of the associated edges with the vertices in R_{i+1} is blocked. Therefore, the main walk may use any of the three vertices in R_{i+1} to visit the clause gadget.

Consider that the main walk must use both single-vertex imprecise regions R_i and R_{i+2}; the only difference between the above cases is using the vertices in multiple-vertex imprecise region R_{i+1}. Therefore, there will be a valid solution to Problem 2, if at least one of the literals is *true*. □

4.3 Transmission Gadget

To complete our 3-SAT instance, we need to connect all of the neighbour scissor gadgets together to make chains, use junction gadgets where we need to copy the truth value of the chains, and use clause gadgets to connect three chains of scissor gadgets to each other. By doing this, we make several walks (because of the holes in the

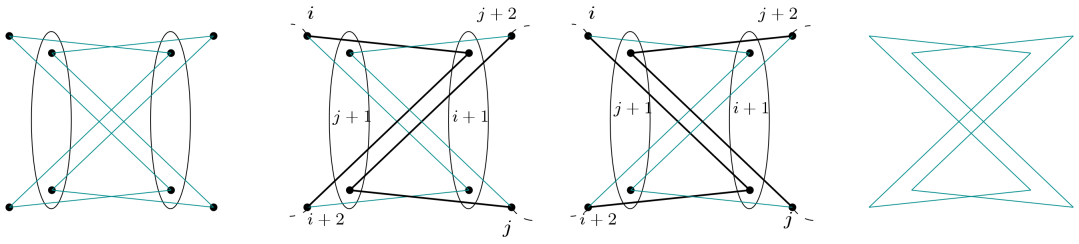


Figure 4.9: (a) A transmission gadget. (b) One of the solutions, representing the *True* state. (c) The other solution, representing the *False* state. (d) Schematic representation for a transmission gadget. To being more readable, we just write i instead of R_i and j instead of R_j, and so on to show the imprecise regions.

construction), while we need only *one* walk through our instance. Before adding the transmission gadgets, the resulting set of closed walks corresponds to a polygon with holes. The transmission gadget removes the holes to form a single simple polygon, i.e., a weakly simple walk in G.

Let B denotes the set of transmission gadgets and $b_1, \ldots, b_k$ represent each transmission gadget, where $b_i \in B$. A *transmission gadget* is a construction of two dual-vertex imprecise regions (each of which has one vertex on the top and one on the bottom) and four single-vertex imprecise regions. In this construction *two* pieces of chain visit the gadget, one from top and the other one from bottom. We show the order of imprecise regions with i for the chain that visits the transmission gadget from top and j for the chain that visits the transmission gadget from bottom. We describe coordinates for a single transmission gadget; all other transmission gadgets are translated copies of a single transmission gadget. The following shows the coordinate of a transmission gadget components:

1: A disc-shaped single-vertex imprecise region $(i + 3)$ with its vertex at (1, 0) and radius $r = \epsilon$, point center $c = (1, 0)$.

2: A disc-shaped dual-vertex imprecise region $(i+1, j+1)$ with its vertices at (9,

2), (9, 4) and radius $r = 2+\epsilon$, point center $c = (9, 3)$.

3: A disc-shaped single-vertex imprecise region (i) with its vertex at (1, 6) and radius $r = \epsilon$, point center $c = (1, 6)$.

4: A disc-shaped single-vertex imprecise region $(j + 3)$ with its vertex at (12, 0) and radius $r = \epsilon$, point center $c = (12, 0)$.

5: A disc-shaped dual-vertex imprecise region $(i + 2, j + 2)$ with its vertices at (4, 2), (4, 4) and radius $r = 2+\epsilon$, point center $c = (4, 3)$.

6: A disc-shaped single-vertex imprecise region (j) with its vertex at (12, 6) and radius $r = \epsilon$, point center $c = (12, 6)$.

A transmission gadget has eight edges. Two of them connect single-vertex imprecise region R_i to the top and bottom vertices of imprecise region R_{i+1}. The other two edges connect single-vertex imprecise region R_{j+2} to the top and bottom vertex of imprecise region R_{j+1}. Two edges connect single-vertex imprecise region R_{i+2} to the top and bottom vertices of imprecise region R_{i+1} and finally the last two edges connect single vertex imprecise region R_j to the top and bottom vertex of imprecise region R_{j+1}.

Lemma 4.5. *There are only two possible ways to make a weakly simple walk through a transmission gadget.*

Proof. In a transmission gadget, two chains enter the gadget, one from top single-vertex imprecise region R_i and the other one from bottom single-vertex imprecise region R_j. The walk has two possible edges to follows out of R_i and R_j, each leading to R_{i+1} and R_{j+1}. Each vertex in R_{i+1} and R_{j+1} has degree two, meaning that the walk must exit that vertex on the second edge. One possible solution is that the first chain starts from the top leftmost single-vertex imprecise region R_i, then uses the top

vertex of the dual-vertex imprecise region R_{i+1}, and then visits the bottom leftmost single-vertex imprecise region R_{i+2}. On the other hand, the other chain visits the transmission gadget by using the single-vertex imprecise region R_j and follows out of R_j to the bottom vertex of the dual-vertex imprecise region R_{j+1} and then visits top rightmost single-vertex imprecise region R_{j+2}. We consider this solution as the *true* state. As described above, each vertex in the transmission gadget has degree two. In this case, the other possible solution is that the first chain uses the top vertex in R_{i+1} and second chain uses the top vertex of R_{j+1}; in this case, when the first chain wants to follows out to R_{i+2}, there will be a crossing in the walk. Moreover, if the first chain uses the bottom vertex in R_{i+1} and the second chain uses the bottom vertex in R_{j+1}, then there will be a crossing when the second chain wants to follows out to R_{j+2}. Therefore, there is only one possible solution for the *true* state; see Figure 4.9(b).
The other solution is that the first chain starts from the top leftmost single-vertex imprecise region R_i, then uses the bottom vertex of the dual-vertex imprecise region R_{i+1}, and then visits the bottom leftmost single-vertex imprecise region R_{i+2}. Also, the other chain visits the transmission gadget by using the single-vertex imprecise region R_j and then follows out of R_j to the bottom vertex of the dual-vertex imprecise region R_{j+1} and then visits top rightmost single-vertex imprecise region R_{j+2}. We consider this solution as the *false* state. Because of the degree two for all of the vertices in the transmission gadget, the other possible solution is that the first chain uses the top vertex of R_{i+1} and second chain uses the top vertex of R_{j+1}; but when the second chain wants to follows out to R_{j+2}, there will be crossing in the walk. Moreover, if the first chain uses the bottom vertex in R_{i+1} and the second chain uses the bottom vertex in R_{j+1}, then there will be a crossing when the first chain wants

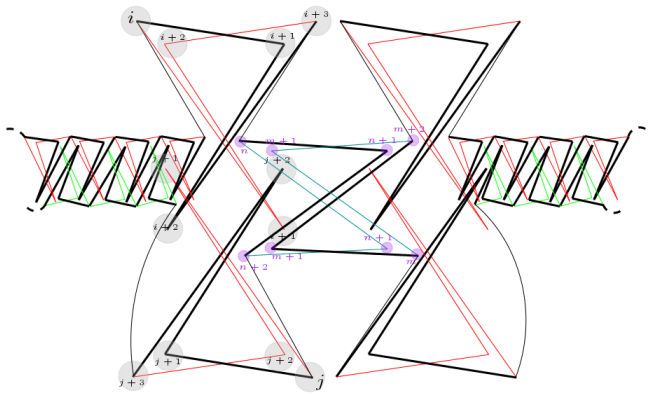
to follows out of R_{i+1} to R_{i+2}. Therefore, there is only one possible solution for the *false* state; see Figure 4.9(c).

If we choose any other vertex set instead of these two possible solutions, there will be crossing in the walk. □

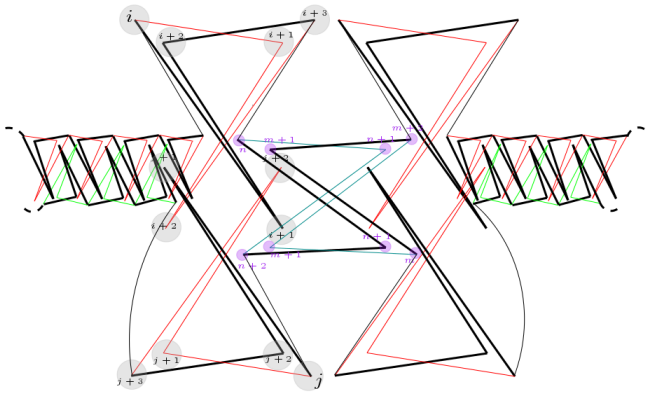
Because a transmission gadget does not have positive or negative legs, we need to link a transmission gadget to two junction gadgets on its sides. So, we use two junction gadgets to connect the transmission gadget to the chain of scissor gadgets, see Figure 4.10. By doing this, we guarantee that the whole chain uses either positive or negative legs (i.e., all of the gadgets are in the same state). Now we can connect all of the unconnected chains together to have just one walk along our construction.

4.4 Final Construction

As described above, the bipartite graph associated with each instance I of PLANAR 3-SAT has a non-crossing drawing in the plane such that all variable vertices are drawn on a line. Our reduction replaces edges with wires consisting of a sequence of scissor gadgets, along with junction gadgets to copy their truth values, and clause gadgets. We require a *single* weakly simple walk in the resulting graph. Cycles in the drawing of I result in disconnected loops in our walk. Using transmission gadgets, we can introduce breaks in the cycles of the drawing of I such that a weakly simple walk through the imprecise regions of the resulting graph bounds a polygonal region whose interior is tree-shaped. To do so, we need the truth value of a wire to be transmitted across the break in the wire, even though the walk does not cross the break. It suffices



(a)



(b)

Figure 4.10: (a) A transmission gadget and its connection with junction gadgets in *True* state. (b) A transmission gadget and its connection with junction gadgets in *False* state. The grey circles and black numbers on them represent the imprecise region of junction gadget and the purple circles and the purple numbers on them denote the imprecise regions of the transmission gadget. To make the figure more clear, instead of drawing the whole imprecise regions, we only show each vertex of the imprecise regions. For junction gadget, the imprecise regions are indicated by a gray circle and the order of the them are denoted by black numbers. For transmission gadget, the imprecise regions are indicated by a purple circle and the order of the them are denoted by purple numbers.

to introduce exactly one transmission gadget per face in the planar drawing of I.

Now that we have all of the gadgets we can replace them in the PLANAR 3-SAT instance (Figure 4.1(c)) and make our final construction. Consider that to replace the edges of the construction, we can easily bend a chain of scissor gadget to any

degree and replace the edge of the PLANAR 3-SAT instance with the bent chain. See Figure 4.4.

Lemma 4.6. *The instance of PLANAR 3-SAT is satisfiable if and only if our transformed instance has a weakly simple walk visiting the sequence of imprecise regions in order.*

Proof. Case 1. Suppose the instance of PLANAR 3-SAT has a satisfying truth assignment. In this case, Lemma 4.1 proves that there is a weakly simple walk that visits a scissor gadget with two possible solutions (i.e., there are two valid solution for a scissor gadget to be visited). Lemma 4.2 proves that there are only two possible ways for two or more scissor gadgets to be linked, corresponding to the transmission of the truth value along all scissor gadgets linked in sequence to for a wire. Also, Lemma 4.3 proves that if we want link wires to a junction gadget, all of the gadgets should be in the same state. Moreover, for having a valid input to a clause gadget, Lemma 4.4 proves that at least one of the literals must be in the true state to have a weakly simple path through our instance. Finally, Lemma 4.5 proves that a transmission gadget must be visited with one of two possible solutions to have a weakly simple walk. When there is a valid solution to all of the gadgets, there is a valid solution to Problem 2. Therefore, we can build our construction with a weakly simple walk which can visit all of the gadgets in order without crossing and provide a satisfiable input to our PLANAR 3-SAT instance.

Case 2. Suppose the instance of PLANAR 3-SAT does not have a satisfying assignment. In this case, for every truth assignment at least one clause is not satisfied. In Lemma 4.1 and Lemma 4.2 if the walk does not use one of the possible solutions, there is a crossing in the walk. Therefore, there is no possible solution to the problem.

Moreover, if the wires are not in the same state when they are linked to a junction gadget (Lemma 4.3), there is no satisfiable solution to the Problem 2. Lemma 4.4 proves that there is a case (where all of the literals are false) that there is no weakly simple walk that visits the sequence of imprecise regions. Also, in a transmission gadget if the walk does not use one of the valid states in Lemma 4.5 there will not be any satisfiable input to the problem. If any of the above cases occurs, then either there is a crossing in the walk, or the walk is disconnected, or the walk does not visit all of the imprecise regions in the specified order. When there is not a valid solution to at least one the gadgets, there is no valid solution to Problem 2. The above cases are contradictions and they should not happen; otherwise, there is no solution to Problem 2.

□

Lemma 4.6 proves the following theorem:

Theorem 4.7. *Given a set of sequenced imprecise regions, the problem of finding a weakly simple walk that visits the imprecise regions in order, is NP-hard.*

Chapter 5

Find a Weakly Simple Walk in a Graph Containing a Non-Crossing Spanning Tree

5.1 Algorithm Description

In this chapter, we seek to find a weakly simple walk when there exist a non-crossing spanning tree for the input graph G (see Problem 3 in Section 1.2.2). As previous literature proposed [43, 45] it is NP-complete to find a non-crossing spanning tree for a given geometric graph. We propose an algorithm to find a weakly simple walk (not necessarily of minimum length) for a given graph G drawn in the plane, with a given non-crossing spanning tree $T \subseteq G$, and a sequence of imprecise regions $R_1, \ldots, R_k$ such that the walk visits the regions in the specified order when crossings are not allowed to be included in the walk. The results in this chapter apply to arbitrary imprecise regions, i.e., when each imprecise region R_i is an arbitrary subset

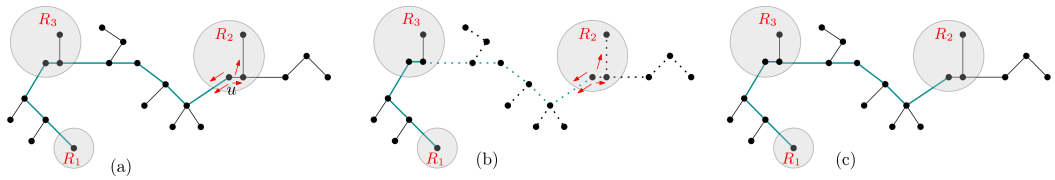


Figure 5.1: This figure illustrates an example of the second round of the algorithm. In the previous round, the walk ended at vertex u, which was the first vertex of R_2 encountered. The partial solution constructed so far consists of a path from the start vertex in R_1 to u; these edges are visited edges of T form the subtree T^*, highlighted in cyan. In the current round, the algorithm searches along the tree T, starting from u. The algorithm explores the graph in a manner similar to depth-first search, moving outward from u. When moving along T^*, the search splits into two, exploring the left and right sides separately, since the solution walk cannot cross itself. The first vertex encountered by the search is v, walking along the top (counterclockwise) side of T^*. This walk from u to v is added to the partial solution at the end of the second round, and any of its edges not yet in T^* are added to T^*. The dotted edges of T in Figure 5.1(b) are explored during the second phase. The edges of T^* at the conclusion of the second round are highlighted in cyan in Figure (c).

of $V(G)$. Consequently, the results apply when imprecise regions are discs in the plane. Although the algorithm does not return an optimal solution in terms of minimizing the total length of the walk, a secondary goal remains to find a weakly simple walk whose total length is short. This algorithm can be implemented on any arbitrary non-negative weighted graph G drawn in $\mathbb{R}^2$ as long as there exists a non-crossing spanning tree T of G. Our algorithm assumes T is given with the input. The algorithm runs in $O(kn)$ time, where k denotes the number of imprecise regions and n denotes the number of vertices in G.

Let $G = (V(G), E(G))$ be any arbitrary graph drawn in the plane, where $V(G)$ denotes the vertex set of G and $E(G)$ represents the edge set of G and each imprecise region R_i is any arbitrary region in the plane (e.g., a disc) . Let $T = (V(T), E(T))$ be a *non-crossing spanning tree* of G, where $V(T) = V(G)$ and $E(T) \subseteq E(G)$.

Instance: A non-negative weighted graph G drawn in the plane, a non-crossing

spanning tree T of G, and a sequence $R_1, \ldots, R_k$ of imprecise regions in the plane.

Problem: Finding a *weakly simple walk* in the graph G (not necessarily of minimum length) that starts at some vertex $v_1 \in V(G) \cap R_1$, follows a sequence of adjacent vertices from v_1 to some vertex $v_2 \in V(G) \cap R_2$, and so on, until reaching some vertex $v_k \in V(G) \cap R_k$.

The easiest solution to this problem is to start from any vertex $v_1 \in V(G) \cap R_1$ to perform a non-crossing tree traversal of T by walking along the edges of T in the plane (i.e., following the outer face) in clockwise or counterclockwise order until returning to v again. If we repeat this process k times we can guarantee that all of the imprecise regions will be visited in the given order. Obviously, the running time for this naive algorithm is $O(kn)$ time and The number of edges of the walk in T^* will be at most kn.

Ideally, we try to find a *short* walk, but the solution returned will not be a shortest

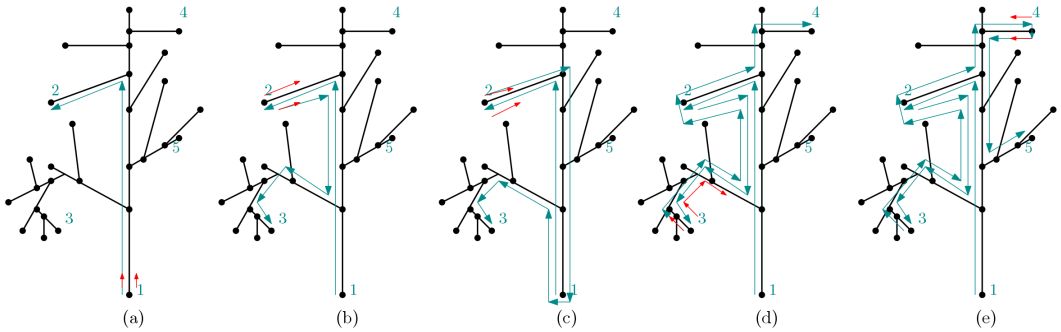


Figure 5.2: An example to show how the algorithm explores the left and right sides of T^*. Black edges show the non-crossing tree T, the cyan lines show a weakly simple walk (or subtree T^*) and numbers denote the order of imprecise regions. The red arrows show two possible solution in each step. (a) Finding a weakly simple walk from v_1 to v_2. (b) Finding a weakly simple walk from v_2 to v_3. (c) Shows the other possible solution for going from v_2 to v_3 which is not shorter than (b) (this solution is not acceptable). (d) Finding a weakly simple walk from v_3 to v_4. (e) Finding a weakly simple walk from v_4 to v_5. To simplify this example, each imprecise region consists of a single vertex.

walk in general. To solve the problem more efficiently (find a shorter walk than that returned by the naive algorithm above), we introduce Algorithm 3. In the first, called *findStartVertex*, we find the closest pair of vertices $u \in R_1 \cap V(T)$ and $v \in R_2 \cap V(T)$. This could be achieved by computing all-pairs shortest distances on T. A more efficient approach is to apply a variant of Breadth-First Search (BFS), using a priority queue instead of a queue since T is a Euclidean graph. Each element's priority in the queue corresponds to its distance from the nearest vertex in R_1. Initially, each vertex $u \in R_1$ is added to the priority queue with priority 0 (its distance from some vertex in R_1). The next vertex v in the BFS is retrieved by *extractMin*. Each neighbour w of v along an unvisited edge (v, w) is added to the priority queue, with corresponding priority set as the priority of v plus the length of the edge (v, w). A flag is set to signal that the edge (u, v) has been visited. Additionally, each element inserted into

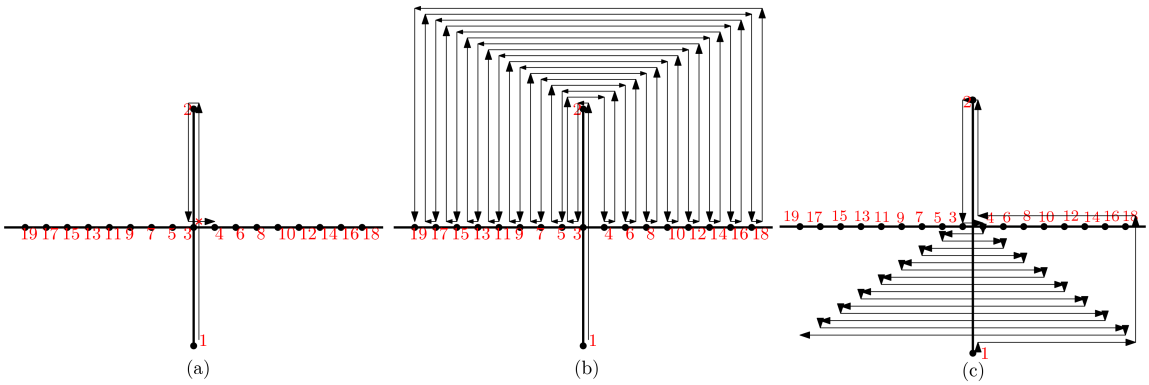


Figure 5.3: Another example for finding a weakly simple walk through a non-crossing spanning tree, when the walk visits the imprecise regions in the given order (a) Represents an invalid solution to the problem (because of the crossing in the walk); the red cross shows a crossing in the walk. (b) A valid solution to the problem by using Algorithm 3. The walk starts from v_1, goes to v_2, ..., and finally visits v_{19}. (c) This is an example which provides a shorter weakly simple walk compare to (b) without using Algorithm 3. Consider that the walk is exactly near the edges in the tree and we draw the walk with a distance to make it more clear (i.e., this distance is ϵ in real case).

the priority queue records the vertex in R_1 nearest to it; this allows a vertex in R_1 nearest to some vertex in R_2 to be returned. The algorithm uses a priority queue whose elements are ordered pairs with a priority, each consisting of a vertex u in $V(T)$ and the vertex v in R_1 nearest to u, with the corresponding priority set as the distance in T from v to u.

Algorithm 2 Finding a nearest pair $\{u, v\}$, where $u \in V(T) \cap R_i$ and $v \in V(T) \cap R_{i+1}$

```
Func findStartVertex(T):

min ← ∞

startV ← any vertex u in R[1]

pq ← empty priority queue

for all edges (u, v) in E(T) do
| (u, v).visited ← false
    end

for all u in R[1] do
| pq.insert((u, u), 0)
    end

while !pq.isEmpty() do
| ((v, u), priority) ← pq.extractMin()
| if v in R[2] AND priority < min then
| |  min ← priority
| |
| |  startV ← u
| |   end
| else
| |  for all neighours w of v do
| |  |  if !(v, w).visited then
| |  |  |  (v, w).visited ← true
| |  |  |
| |  |  |  q.insert((w, u), priority + weight(v, w))
| |  |  |   end
| |  |  end
| |   end
|    end

Return startV
```

Each edge is visited at most once in each direction, giving $O(n)$ time per iteration (i.e., per imprecise region).

After finding the start vertex with *findStartVertex* (Algorithm 2), our algorithm proceeds in rounds to find a shortest path in T from the last vertex visited in $R_i \cap V(T)$ to the nearest vertex in $R_{i+1} \cap V(T)$, without crossing any edges of T previously visited (Algorithm 5). We consider the visited edges as a subtree of T which we call T^*. The walk may traverse on the edges of T (Algorithm 5) or it may traverse on the right side or left side of T^* edges (Algorithm 4), because the walk cannot include a crossing. We use two different functions to traverse T (Algorithm 5) and both sides of T^* (Algorithm 4) (clockwise and counterclockwise) and return each corresponding walk and its length which the walk traversed. The rest of the algorithm (Algorithm 3) proceeds in $k-2$ rounds (i.e., the algorithm is repeated k times and we calculate the first and the second run separately), where each round adds a path from the vertex in R_{i-1} at which the previous round ended, to the nearest vertex in R_i. The algorithm uses a Depth First Search (DFS) algorithm when it traverses T after the first step (Algorithm 4). The algorithm explores the graph by walking around T^* in clockwise and counterclockwise directions (Algorithm 4), as well as along edges of T not in T^* (Algorithm 5). The two searches proceed in opposite directions around T^*. Any branches of T that lie between the last edge of T^* traversed and the next clockwise (respectively, counterclockwise) edge of T^* are explored recursively, before continuing the search clockwise (respectively, counterclockwise) around T^*. Each recursive search returns the path to the nearest vertex of R_i and the corresponding path length, or ∞ if no vertex of R_i is found. The shortest path found is added to the solution walk at the end of the round. For the first step, the algorithm tries all

possible paths from the start vertex using a neighbour v as $prevV$, but this will not try the path starting with edge $(startV, v)$, so we need to call function explore again starting at v with $prevV$ set to the start vertex. Figure 5.1 shows an example of how the algorithm works and Figure 5.2 represents an example of how left or right side of the edges is chosen.

Algorithm 3 Finding a non-crossing walk

```
Func findNonCrossingWalk(T, R):

solutionWalk ← empty queue // store solution as a queue of edges

for all edges e in E(T) do
| e.T* ← false // no edges in T* initially
    end

startV ← findStartVertex(T, R)

prevV ← any neighbour of startV

minLength ← ∞

minPath ← empty double-ended queue

// In the first round, T* is empty

(path, pathLength) ← explore(T, R, 2, startV, prevV)

if pathLength < minLength then
| minLength ← pathLength
|
| minPath ← path
    end

(path, pathLength) ← explore(T, R, 2, prevV, startV)

path.addToHead((startV, prevV))

pathLength += length(startV, prevV)

if pathLength < minLength then
| minLength ← pathLength
|
| minPath ← path
    end

// Add the path found in the first round to the solution.

while !minPath.isEmpty() do
| nextEdge ← minPath.extractHead()
|
| nextEdge.T* ← true // add edge to T*
|
| solutionWalk.enqueue(nextEdge) // add edge to solution walk
    end
```

startV ← nextEdge.tailVertex

prevV ← nextEdge.headVertex

dir ← cw

// remaining rounds

for *i = 3 to k* **do**

if *dir = cw* **then**
firstFace ← (prevV, startV, startV.cwT^*(prevV)
end

else
firstFace ← (prevV, startV, startV.ccwT^*(prevV)
end

minLength ← ∞

minPath ← empty double-ended queue

(path, pathLength) ← exploreT^*(T, R, i, firstFace, startV, prevV, cw)

path.addToHead((prevV, startV))

pathLength += length(startV, prevV)

if *pathLength < minLength* **then**
minLength ← pathLength
minPath ← path
dir ← cw
end

(path, pathLength) ← exploreT^*(T, R, i, firstFace, startV, prevV, ccw)

path.addToHead((prevV, startV))

pathLength += length(startV, prevV)

if *pathLength < minLength* **then**
minLength ← pathLength
minPath ← path
dir ← ccw
end

while *!minPath.isEmpty()* **do**
  nextEdge ← minPath.extractHead()
  nextEdge.T^* ← true // add edge to T^*
  solutionWalk.enqueue(nextEdge) // add edge to solution walk
  end
startV ← nextEdge.tailVertex
prevV ← nextEdge.headVertex **end**
Return solutionWalk

Algorithm 4 Explore T^*

```
// exploreT*: start at the vertex startV on T*, having come from its neighbour
//prevV in the direction dir, cw (clockwise) or ccw (counterclockwise), explore
//T* in the direction dir, and explore any branches of T outside T* that are
//encountered. Return the shortest path from startV to the closest vertex in R_i.
// startV - start vertex
// prevV - last vertex visited before startV
// firstFace - face at which the current round started its search. A face is defined
//by three consecutive vertices, i.e., two edges, on the boundary of the face.
//cw and ccw are two functions that traverse T* clockwise and counter-clockwise.
Func exploreT*(T, R, i, firstFace, startV, prevV, dir):
minLength ← ∞
minPath ← empty double-ended queue
if dir = cw then
| currentFace ← (prevV, startV, startV.cwT*(prevV)
    end
else
| currentFace ← (prevV, startV, startV.ccwT*(prevV)
    end
if startV in R[i] then
| // found a vertex in the imprecise region R_i
| minLength ← 0
    end
else
| if dir = cw then
| | u ← startV.cw(prevV)
|  end
| else
| | u ← startV.ccw(prevV)
|  end
    end
```

```
while !(startV, u).T* do
    // explore edges of T outside T*
    (path, pathLength) ← explore(T, R, i, u, startV)
    path.addToHead((startV, u))
    pathLength += length(startV, u)
    end
if pathLength < minLength then
    minLength ← pathLength
    minPath ← path
    end
if dir = cw then
    u ← startV.cw(u)
    end
else
    u ← startV.ccw(u)
    end
// continue exploring T* in direction dir
if currentFace != startFace then
    (path, pathLength) ← exploreT*(T, R, i, firstFace, u, startV, dir)
    path.addToHead((startV, u))
    pathLength += length(startV, u)
    if pathLength < minLength then
        minLength ← pathLength
        minPath ← path
        end
    end
Return (minPath, minLength)
```

Algorithm 5 Explore

```
Func explore(T, R, i, startV, prevV):
    minLength ← ∞
    minPath ← empty double-ended queue
    if startV in R[i] then
        // found a vertex in the imprecise region R_i
        minLength ← 0
    end
    else
        u ← startV.cw(prevV)
        while u != prevV do
            (path, pathLength) ← explore(T, R, i, u, startV)
            path.addToHead((startV, u))
            pathLength += length(startV, u)
            if pathLength < minLength then
                minLength ← pathLength
                minPath ← path
            end
            u ← startV.cw(u)
        end
    end
    Return (minPath, minLength)
```

5.2 Time Analysis

Let k denote the number of imprecise regions and let n denote the number of vertices. Since the input graph is a tree, T, each edge is visited at most once in each direction, giving $O(n)$ time per iteration (i.e., for each imprecise region R_i). In the worst case, Algorithm 3 should traverse all of the n vertices k times. Each step requires only constant time to move to the next vertex of T^* (or T) along the unique route determined by the left walk or right walk around T^*. Therefore, Algorithm 3 can be run in $O(kn)$ time.

Theorem 5.1. *Given a non-negative weighted graph G drawn in the plane, a non-crossing spanning tree T where $T \subseteq G$, and a sequence $R_1, \ldots, R_k$ of imprecise regions in the plane, there exists an algorithm that finds a weakly simple walk which starts at some vertex $v_1 \in V(G) \cap R_1$, follows a sequence of adjacent vertices from v_1 to some vertex $v_2 \in V(G) \cap R_2$, and so on, until reaching some vertex $v_k \in V(G) \cap R_k$, in $O(kn)$ time.*

5.3 Implementation

We implemented Algorithm 3 with C++. We generated an undirected tree T, where the weight of each edge (u, v) corresponds to the Euclidean distance between u and v. We choose random coordinates (i.e., x and y) for each vertex of the tree T where they can be any *double* between $[0, 1)$. To generate this tree, first we randomly generate two vertices and we connect them. Then, we randomly generate the third vertex and connect this vertex randomly to one of the previous vertices. Obviously, there is no crossing for these two steps. As the next step, we randomly generate one

more vertex and connect this vertex randomly to one of the previous vertices. Now, we check whether there is a crossing with the previous edges or not. If there is a crossing, we eliminate the last generated vertex and we generate another random vertex and we do this process again to generate our tree. We consider 25, 50, 100, 200, ..., 1000, 1500, 2000, 2500, 3000, and 3500 as the number of vertices for the tree T and we choose 10%, 25%, 50%, and 75% of the number of vertices as the number of imprecise regions (which we call the *imprecise region density*). For example, we choose 500 as the number of vertices and we run the code for 50, 125, 250, and 375 imprecise region density. We repeated this process 100 times for each number of vertices with different imprecise region density. We run our code on a Microsoft Surface Pro with 7th gen Intel Core i5 processor, 8GB of ram and 256GB SSD. The execution time for the couple first number of vertices with different imprecise region density was pretty fast (e.g., less than 5 minute) while for 3500 vertices with 75% imprecise region density it took near 35 hours to get the results. Due to the lack of powerful hardware, we

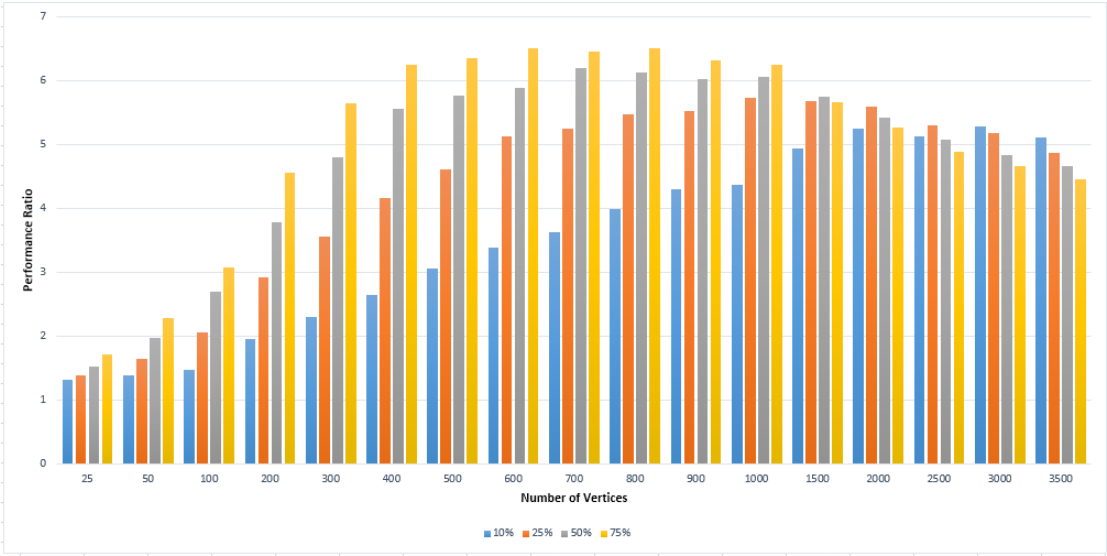


Figure 5.4: Final results of Algorithm 3. This bar chart shows the performance ratio based on the number of vertices.

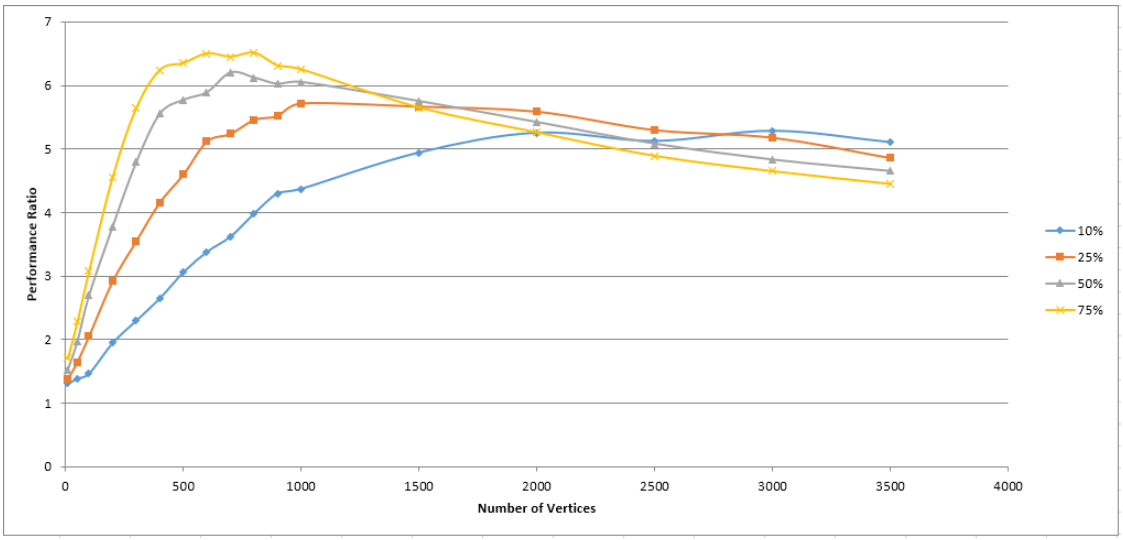


Figure 5.5: The linear representation of the results of Algorithm 3.

could not run our algorithm for more vertices. In this experiment, each imprecise region contains exactly one vertex. We stored the total weight of the returned walk of Algorithm 3 in *totalWeight* which is a variable that stores the Euclidean length of the returned walk. Since we do not have an algorithm for computing the length of the optimal weakly simple walk, to evaluate the performance of our algorithm we introduce a lower bound on the length of the optimal weakly simple walk. Obviously, the total weight of the walk in the best case cannot be less than the sum of the shortest walk between each pair of imprecise regions. Therefore, we consider it as a lower bound for our algorithm. We calculated and stored the weight of the shortest walk between the regions in *lowerBound*. Finally, we divided *totalWeight* to *lowerBound* to evaluate the *performance ratio* for each instance on which our algorithm was run.

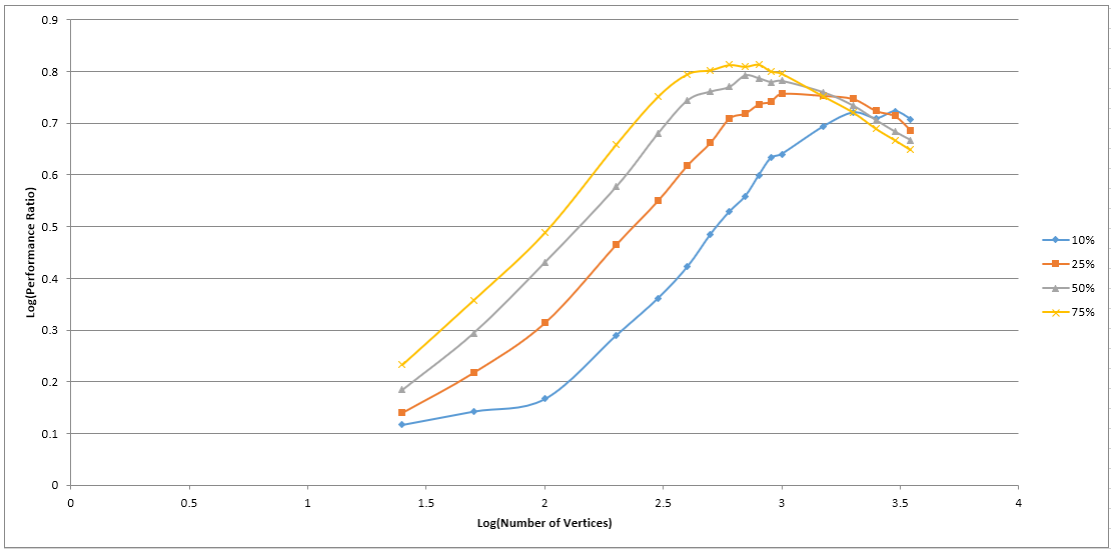


Figure 5.6: Log-log plot of the results of Algorithm 3

5.4 Result

We show the final results of the performance ratio of Algorithm 3 in Figure 5.4. As can be seen, for 25 vertices, the performance ratio is less than 2 for different imprecise region densities. When the number of vertices is between 25 and 800, the performance ratio increases as n increases and as the imprecise region density increases. However, after 800 vertices, the performance ratio shows a different behaviour; for larger imprecise region density, the performance ratio starts to decrease. Moreover, imprecise regions with the lower density follow this behaviour after a certain number of vertices. It seems that after a certain number of vertices, the performance ratio will decrease for all of the different imprecise region densities and the final plot will be a bell-shape plot. Also, it seems the performance ratio has an upper bound and it will never goes more than 7. The overview in Figure 5.5 shows a linear representation of the final results. Figure 5.6 shows a log-log plot for the results where x-axis represents the logarithm of the number of imprecise regions and y-axis denotes the logarithm of the

performance ratio. As it is seen, when the number of vertices is between 25 and 800 the performance ratio for any given imprecise region density is proportional to n^c where c is a constant. The fact that the slope appears to remain unchanged (i.e., all four plots appear parallel) suggest that c is independent of the density of imprecise regions to n. Since the plots appear linear when the number of vertices is between 25 and 800 in Figure 5.6, this suggest $c \approx 1$. Also, Figure 5.6 shows that the log-log plot will decrease and it seems it will be a bell-shape plot after a certain number of vertices.

5.5 Discussion

In this section, we mostly focused on finding a weakly simple walk and tried to minimize the length of the walk. For example, Figure 5.3(b) and (c) show two examples of different algorithms applied on a the same non-crossing spanning tree. Applying Algorithm 3 to the non-crossing spanning tree, results in Figure 5.3 (b); while it is not a shortest path (although it is weakly simple) we can often get a better result such as Figure 5.3 (c) by applying a different algorithm. As can be seen, if we apply an algorithm which chooses a different path to visit $v_2 \in R_2$ at the first step, then we can get better results (i.e., a shorter weakly simple walk).

It is still an open problem whether the approximation ratio of Algorithm 3 can be bounded. Also, finding an algorithm (such as what is applied on Figure 5.3 (c)) that finds a better solution which is both *weakly simple* and *shortest* or at least *shorter* than what we proposed is an open problem.

Chapter 6

Conclusion and Open Problems

In Chapter 3, we introduced an exact algorithm to find a walk of minimum length in G (where G can be any arbitrary graph drawn in the plane) which visits the imprecise regions $R_1, \ldots, R_k$ in the specified order when the walk may include crossings (see Section 1.2.2). We introduced Algorithm 1 for this purpose which runs in $O(kn^2 + APSP(n, m))$ time, where $APSP(n, m)$ denotes the time required to compute an $[n \times n]$ all-pairs shortest distances matrix for G. When the imprecise regions are mutually disjoint, the running time reduces to $O(n^2 + APSP(n, m))$ which, for most graphs, corresponds to $O(APSP(n, m))$. We decided to use Dijkstra's algorithm to find APSP. Consider that $APSP(n, m) \in O(nm + n^2 \log n)$, therefore the total running time is $O(n^2 \log n)$ for planar graphs. Also, we provided a proof for the time analysis for Algorithm 1 in Chapter 3. In this chapter it remained open whether there exists an algorithm that solves Problem 1 in $f(n, m) \in o(kn^2)$ time for general imprecise regions and $f(n, m) \in o(n^2)$ time when the imprecise regions are mutually disjoint.

In Chapter 4, we solved a special case of Löffler's open problem [51]. We show

that Problem 2 (see Section 1.2.2) is NP-hard, even when the length of the walk is not required to be minimized. We considered a sequence S of k imprecise regions (possibly overlapping) in an Euclidean graph G and we sought a weakly simple walk that starts at some vertex $v_1 \in V(G) \cap R_1$, follows a sequence of adjacent vertices from v_1 to some vertex $v_2 \in V(G) \cap R_2$, and so on, until reaching some vertex $v_k \in V(G) \cap R_k$. We described a reduction that transforms an instance of PLANAR 3-SAT into a graph drawn in the plane and a sequence of disc-shaped imprecise regions to show that it is NP-hard to decide whether a weakly simple walk exists.

In Chapter 5, we sought to find a weakly simple walk (not necessarily of minimum length) when there exists a non-crossing spanning tree for the graph G (see Section 1.2.2, Problem 3). Based on previous literature, it is NP-complete to find a non-crossing spanning tree for a geometric graph. Therefore, we proposed an algorithm which can be applied on any non-negative weighted graph G drawn in $\mathbb{R}^2$ when given a non-crossing spanning tree T of G. The purpose of Algorithm 3 was to find a weakly simple walk (not necessarily of minimum length) which visits the imprecise regions $R_1, \ldots, R_k$ in the specified order when crossing is not allowed to be included in the walk. The running time of Algorithm 3 is $O(kn)$ time. We implemented the algorithm with C++ and we analyzed experimental results for various graph sizes and varying densities of imprecise regions. We conclude that the performance ratio appears to decrease after a certain number of vertices and also it appears that when the number of vertices is between 25 and 800 the performance ratio is proportional to n^c, where $c \approx 1$. Also, based on the results, we proposed that c is independent of the ratio of imprecise regions to n when the number of vertices is between 25 and 800. Also, based on the experimental results, we suggest that the upper bound on

the performance ratio for Algorithm 3 appears to be 7. It remains open whether the approximation ratio of Algorithm 3 can be bounded analytically in the worst case, and whether a better algorithm (such as Figure 5.3(c)) which always finds a shorter walk can be found.

Bibliography

[1] https://www.google.com/maps. Accessed: 2019-01-12.

[2] http://ulcrobotics.com/. Accessed: 2019-01-06.

[3] http://crawlerpipeline.com/index.php/our-services/robotic-camera-inspection-cctv. Accessed: 2019-01-06.

[4] http://www.rangedale.com.au/robotics/. Accessed: 2019-01-06.

[5] https://www.cleancosystems.com/pipeline-inspection/. Accessed: 2019-01-06.

[6] https://www.fitbit.com/en-ca/ionic. Accessed: 2019-01-06.

[7] https://buy.garmin.com/en-CA/CA/cIntoSports-cRunning-p1.html. Accessed: 2019-01-12.

[8] https://www.apple.com/ca/ios/health/. Accessed: 2019-01-06.

[9] https://www.strava.com/. Accessed: 2019-01-12.

[10] https://www.endomondo.com/. Accessed: 2019-01-12.

[11] Dso water sewer map. http://gisrevprxy.seattle.gov/wab_ext/DSOResearch_Ext/. Accessed: 2019-01-06.

[12] A. Arampatzis, M. Van Kreveld, I. Reinbacher, C. B. Jones, S. Vaid, P. Clough, H. Joho, and M. Sanderson. Web-based delineation of imprecise regions. *Computers, Environment and Urban Systems*, 30(4):436–459, 2006.

[13] S. Brakatsoulas, D. Pfoser, R. Salas, and C. Wenk. On map-matching vehicle tracking data. In *Proceedings of the 31st international conference on Very large data bases*, pages 853–864. VLDB Endowment, 2005.

[14] T. M. Chan. All-pairs shortest paths with real weights in o (n 3/log n) time. *Algorithmica*, 50(2):236–243, 2008.

[15] T. M. Chan. More algorithms for all-pairs shortest paths in weighted graphs. *SIAM Journal on Computing*, 39(5):2075–2089, 2010.

[16] H. Chang, J. Erickson, and C. Xu. Detecting weakly simple polygons. In *Proceedings of the Twenty-Sixth Annual ACM-SIAM Symposium on Discrete Algorithms, SODA 2015, San Diego, CA, USA, January 4-6, 2015*, pages 1655–1670, 2015.

[17] B. V. Cherkassky, A. V. Goldberg, and T. Radzik. Shortest paths algorithms: Theory and experimental evaluation. *Mathematical programming*, 73(2):129–174, 1996.

[18] P. Clough, H. Joho, C. B. Jones, and R. Purves. Modelling vague places with knowledge from the web. *Unpublished, available at http://ext. dcs. shef. ac. uk/ u0015/darwinProjectProposals/SandersonMark/clough. pdf*, 2005.

[19] A. G. Cohn and N. M. Gotts. The 'egg-yolk'representation of regions with inde-

terminate boundaries. *Geographic objects with indeterminate boundaries*, 2:171–187, 1996.

[20] T. H. Cormen. Section 24.3: Dijkstra's algorithm. *Introduction to algorithms*, pages 595–601, 2001.

[21] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein. *Introduction to algorithms*. MIT press, 2009.

[22] G. Dantzig. *Linear programming and extensions*. Princeton university press, 2016.

[23] C. Davies, I. Holt, J. Green, J. Harding, and L. Diamond. User needs and implications for modelling vague named places. *Spatial Cognition & Computation*, 9(3):174–194, 2009.

[24] M. de Berg, W. Meulemans, and B. Speckmann. Delineating imprecise regions via shortest-path graphs. In *Proceedings of the 19th ACM SIGSPATIAL international conference on advances in geographic information systems*, pages 271–280. ACM, 2011.

[25] E. V. Denardo and B. L. Fox. Shortest-route methods: 1. reaching, pruning, and buckets. *Operations Research*, 27(1):161–186, 1979.

[26] E. Dijkstra. A note on two problems in connexion with graphs:(numerische mathematik, _1 (1959), p 269-271). 1959.

[27] W. Dobosiewicz. A more efficient algorithm for the min-plus multiplication. *International journal of computer mathematics*, 32(1-2):49–60, 1990.

[28] S. Durocher, C. Gray, and J. A. King. Minimizing the number of arcs linking a permutation of points in the plane. In *CCCG*, 2006.

[29] J. Erickson and A. Nayyeri. Shortest non-crossing walks in the plane. In *Proceedings of the twenty-second annual ACM-SIAM symposium on Discrete Algorithms*, pages 297–308. Society for Industrial and Applied Mathematics, 2011.

[30] M. Erwig and M. Schneider. Vague regions. In *International Symposium on Spatial Databases*, pages 298–320. Springer, 1997.

[31] M. L. Fredman. New bounds on the complexity of the shortest path problem. *SIAM Journal on Computing*, 5(1):83–89, 1976.

[32] M. L. Fredman and R. E. Tarjan. Fibonacci heaps and their uses in improved network optimization algorithms. *Journal of the ACM (JACM)*, 34(3):596–615, 1987.

[33] G. Gallo and S. Pallottino. Shortest path algorithms. *Annals of operations research*, 13(1):1–79, 1988.

[34] A. V. Goldberg. Shortest path algorithms: Engineering aspects. In *International Symposium on Algorithms and Computation*, pages 502–513. Springer, 2001.

[35] A. V. Goldberg. A simple shortest path algorithm with linear average time. In *European Symposium on Algorithms*, pages 230–241. Springer, 2001.

[36] A. V. Goldberg and C. Silverstein. Implementations of dijkstra's algorithm based on multi-level buckets. In *Network optimization*, pages 292–327. Springer, 1997.

[37] C. Grothe and J. Schaab. Automated footprint generation from geotags with kernel density estimation and support vector machines. *Spatial Cognition & Computation*, 9(3):195–211, 2009.

[38] H. W. Guesgen. From the egg-yolk to the scrambled-egg theory. In *FLAIRS Conference*, pages 476–480, 2002.

[39] Y. Han. Improved algorithm for all pairs shortest paths. *Information Processing Letters*, 91(5):245–250, 2004.

[40] Y. Han. An o (n 3 (log log n/log n) 5/4) time algorithm for all pairs shortest path. *Algorithmica*, 51(4):428–434, 2008.

[41] R. Jacob, M. Marathe, and K. Nagel. A computational study of routing algorithms for realistic transportation networks. *Journal of Experimental Algorithmics (JEA)*, 4:6, 1999.

[42] G. M. Jacquez, S. Maruca, and M.-J. Fortin. From fields to objects: a review of geographic boundary analysis. *Journal of Geographical Systems*, 2(3):221–241, 2000.

[43] K. Jansen and G. J. Woeginger. The complexity of detecting crossingfree configurations in the plane. *BIT Numerical Mathematics*, 33(4):580–595, 1993.

[44] A. Javanmard, M. Haridasan, and L. Zhang. Multi-track map matching. In *Proceedings of the 20th International Conference on Advances in Geographic Information Systems*, pages 394–397. ACM, 2012.

[45] N. Katoh and S.-I. Tanigawa. Fast enumeration algorithms for non-crossing geometric graphs. *Discrete & Computational Geometry*, 42(3):443–468, 2009.

[46] C. Knauer, M. Löffler, M. Scherfenberg, and T. Wolle. The directed hausdorff distance between imprecise point sets. *Theoretical Computer Science*, 412(32):4173–4186, 2011.

[47] D. E. Knuth and A. Raghunathan. The problem of compatible representatives. *SIAM Journal on Discrete Mathematics*, 5(3):422–427, 1992.

[48] D. Lee, C. Shen, C. Yang, and C. Wong. Non-crossing paths problems. *Manuscript, Dept. of EECS, Northwestern University*, 1991.

[49] Y. Li, Q. Huang, M. Kerber, L. Zhang, and L. Guibas. Large-scale joint map matching of gps traces. In *Proceedings of the 21st ACM SIGSPATIAL International Conference on Advances in Geographic Information Systems*, pages 214–223. ACM, 2013.

[50] D. Lichtenstein. Planar formulae and their uses. *SIAM journal on computing*, 11(2):329–343, 1982.

[51] M. Löffler. Existence and computation of tours through imprecise points. *International Journal of Computational Geometry & Applications*, 21(01):1–24, 2011.

[52] U. Meyer. Single-source shortest-paths on arbitrary directed graphs in linear average-case time. In *Proceedings of the twelfth annual ACM-SIAM symposium on Discrete algorithms*, pages 797–806. Society for Industrial and Applied Mathematics, 2001.

[53] D. R. Montello, M. F. Goodchild, J. Gottsegen, and P. Fohl. Where's downtown?: Behavioral methods for determining referents of vague spatial queries. *Spatial Cognition & Computation*, 3(2-3):185–204, 2003.

[54] E. Papadopoulou. k-pairs non-crossing shortest paths in a simple polygon. In *International Symposium on Algorithms and Computation*, pages 305–314. Springer, 1996.

[55] R. Purves, P. Clough, and H. Joho. Identifying imprecise regions for geographic information retrieval using the web. In *Proceedings of the 13th Annual GIS Research UK Conference*, pages 313–18, 2005.

[56] I. Reinbacher, M. Benkert, M. Van Kreveld, J. S. Mitchell, and A. Wolff. Delineating boundaries for imprecise regions. In *European Symposium on Algorithms*, pages 143–154. Springer, 2005.

[57] N. S. Savage, S. Nishimura, N. E. Chavez, and X. Yan. Frequent trajectory mining on gps data. In *Proceedings of the 3rd International Workshop on Location and the Web*, page 3. ACM, 2010.

[58] M. Schneider. Modelling spatial objects with undetermined boundaries using the realm/rose approach. *Geographic Objects with Indeterminate Boundaries*, 2:141–152, 1996.

[59] R. Sedgewick and J. S. Vitter. Shortest paths in euclidean graphs. *Algorithmica*, 1(1-4):31–48, 1986.

[60] J.-Y. Takahashi, H. Suzuki, and T. Nishizeki. Algorithms for finding noncrossing paths with minimum total length in plane graphs. *Electronics and Communications in Japan (Part III: Fundamental Electronic Science)*, 78(4):1–15, 1995.

[61] T. Takaoka. A new upper bound on the complexity of the all pairs shortest path problem. *Information Processing Letters*, 43(4):195–199, 1992.

[62] T. Takaoka. A faster algorithm for the all-pairs shortest path problem and its application. In *International Computing and Combinatorics Conference*, pages 278–289. Springer, 2004.

[63] T. Takaoka. An o (n3loglogn/logn) time algorithm for the all-pairs shortest path problem. *Information Processing Letters*, 96(5):155–161, 2005.

[64] G. Taylor and G. Blewitt. *Intelligent positioning: GIS-GPS unification*. Wiley Online Library, 2006.

[65] M. Thorup. Undirected single source shortest paths in linear time. In *Foundations of Computer Science, 1997. Proceedings., 38th Annual Symposium on*, pages 12–21. IEEE, 1997.

[66] Q. Wang, Y.-C. Guo, and X.-W. Shi. A generalized gps algorithm for reducing the bandwidth and profile of a sparse matrix. *Progress In Electromagnetics Research*, 90:121–136, 2009.

[67] C. E. White, D. Bernstein, and A. L. Kornhauser. Some map matching algorithms for personal navigation assistants. *Transportation research part c: emerging technologies*, 8(1-6):91–108, 2000.

[68] J. J.-C. Ying, B.-N. Shi, K.-C. Lan, and V. S. Tseng. Spatial-temporal mining for urban map-matching. In *UrbComp The 3rd international workshop on Urban Computing*, 2014.

[69] R. Yuster. Efficient algorithms on sets of permutations, dominance, and real-weighted apsp. In *Proceedings of the twentieth annual ACM-SIAM symposium*

on Discrete algorithms, pages 950–957. Society for Industrial and Applied Mathematics, 2009.

[70] F. B. Zhan and C. E. Noon. Shortest path algorithms: an evaluation using real road networks. *Transportation science*, 32(1):65–73, 1998.

[71] U. Zwick. A slightly improved sub-cubic algorithm for the all pairs shortest paths problem with real edge lengths. *Algorithmica*, 46(2):181–192, 2006.